# — PARA — INICIANTES

desenvolvendo seu primeiro programa

André Carlucci
Carlos dos Santos

Claudenir Andrade
Rafael Almeida

Ray Carneiro
Renato Haddad

# SOBRE OS AUTORES

**André Carlucci**

Desenvolvedor há 19 anos, co-fundador e ex-CTO da Way2 Tecnologia, Microsoft MVP por 9 anos, Microsoft RD, TDC Rockstar, Intel Software Innovator, Intel Black Belt Software Developer.

Palestrante no Brasil e na Europa em mais de 100 eventos técnicos, mentor de startups e Diretor Global de Software na Kinly em Amsterdam.

**Twitter: @twitter.com/andrecarlucci**
**Linkedin: linkedin.com/in/andrecarlucci**

---

**Carlos dos Santos**

Desenvolvedor com mais de 30 anos de experiência.
Microsoft MVP desde 2008 e Microsoft Regional Director.
Responsável pela área de inovação na CDS Informática.
Palestrante em diversos eventos no Brasil e exterior.

Consultor para empresas sobre Nuvem, DevOps, Desenvolvimento e Performance.

**Twitter: @cdssoftware**
**Linkedin: linkedin.com/in/cdssoftware/**

---

**Claudenir C. Andrade**

Apaixonado por pessoas e tecnologia há 25 anos.
Diretor de Integração Tecnológica e Marketing do Grupo Elgin/Bematech.
MVP Microsoft há 20 anos e Regional Director Microsoft.
Diretor de ISVs na AFRAC e Membro do Conselho MTAC.
Escritor de dois livros de Automação Comercial C# e NFCe.
Diretor de comunidades técnicas ISVs, Software Houses, há 23 anos.

**Twitter: @twitter.com/claudenirandrad**
**Linkedin: linkedin.com/in/claudenirandrade**

---

**Rafael Almeida Santos**

Especialista em engenharia de software no Iti Itaú, Microsoft MVP, Instrutor na desenvolvedor.io, contribuidor de projetos open source, mais de 15 anos de experiência em desenvolvimento de software, palestrante, apaixonado por acesso a dados de alta performance, arquitetura, .NET e Cloud.

**Blog: ralms.net**
**Linkedin: linkedin.com/in/ralmsdeveloper**

---

**Ray Carneiro**

Arquiteto Cloud na Microsoft com mais de 10 anos de experiência em projetos de desenvolvimento de software, migração para nuvem e devops. Ex-Microsoft MVP e palestrante em diversos eventos nacionais e internacionais.

Instrutor na plataforma academiadotnet.

**Blog: rcarneironet.medium.com**
**Linkedin: linkedin.com/in/raycarneiro**

---

**Renato Haddad**

Desenvolvedor e apaixonado por tecnologias.
Microsoft MVP desde 2000.
Consultor independente na área de desenvolvimento de softwares, especialista em Power BI.

Apaixonado por kitefoil, bike, canoa havaiana.

**Blog: https://medium.com/@rehaddad**
**Linkedin: linkedin.com/in/renatohaddad**

O Programa MVP é um reconhecimento direto da Microsoft EUA para profissionais em todo o mundo que possuem influência técnica em comunidades, que são especialistas em tecnologia e que impactam pessoas com paixão por tecnologia. São reconhecidos por sua competência técnica e sua influência na comunidade em seu país.

O Programa Regional Director é um Reconhecimento direto da Microsoft EUA para profissionais que possuem senioridade em sua área de atuação tecnológica, que influenciam empresas e tomadores de decisão. Muitos dos Regional Directors possuem atuação profissional em cadeiras de Conselho, Diretoria e são C-Level em suas empresas ou são empreendedores em tecnologia. Possuem impacto nos negócios, na sociedade e são reconhecidamente líderes na região em que atuam.

**Capa** Agência Hex

**Diagramação** Agência Hex

**Apoio** Elgin Developers Community

2021
Versão Digital

# DEDICATÓRIA

Dedicamos este livro a todos que, de alguma forma, participaram de nossa história como MVPs Microsoft. Que nestes 20 anos de programa MVP no Brasil investiram seu tempo em assistir uma palestra nossa, dedicaram seu tempo em trocar informações conosco e nosso muito obrigado por acreditar em nossa missão de compartilhar conhecimento e impactar a vida das pessoas através da tecnologia. Todos os autores deste livro são MVPs Microsoft e outros além de MVPs, são Regional Director Microsoft. Todos devem muito a comunidade e a Microsoft por acreditar que a tecnologia e o conhecimento transformam vidas, impactam pessoas.

Dedicamos este livro para todos os MVPs leads que nos apoiaram ao longo de toda esta jornada, a todos eles pela liderança no Brasil com o Programa MVP, acreditando em nosso profissionalismo e nos apoiando nos momentos mais difíceis nestes últimos 20 anos. Nossa gratidão em forma de dedicação aos:

Nestor Portillo, que foi líder global do programa MVP. Quantas viagens juntos e quantas ideias colocamos em prática aqui no Brasil com sua liderança global.

Leonardo Tolomelli, nosso primeiro MVP Lead no Brasil. Desco-

briu vários MVPs que estão no programa até hoje, trabalhou arduamente pela construção de uma comunidade técnica forte aqui no Brasil.

Rodolfo Roim, que em sua passagem como MVP Lead trouxe uma bagagem importante com uma visão técnica incrível sobre produtos e tecnologia Microsoft.

J.P. Clementi, que marcou o programa MVP inovando em sua liderança, seu carisma, sua atenção e cuidado com cada um dos MVPs.

Fernanda Saraiva, que oriunda de comunidade, com belo conhecimento técnico, e como também já foi MVP entendeu bem nossas necessidades e liderou o programa de maneira incrível

Glauter Januzzi, hoje nosso atual líder do programa MVP não apenas Brasil, mas também na América Latina, trouxe uma dinâmica diferente e ágil para o programa MVP no Brasil. Glauter vem de comunidade, liderou comunidades, entende as dores e as necessidades da comunidade e acima de tudo valoriza as pessoas, colocando-as antes da tecnologia.

Mary Kate Sincero, sua liderança mundial no programa Regional Director tem feito a diferença. Obrigado por toda a dedicação e por confiar em nós. Seu trabalho árduo e dedicado é inspirador.

Fabio Hara, que todos estes anos na Microsoft sempre está ao nosso lado, nos apoiando, trazendo oportunidades de engajamento para todos os MVPs, sempre pensando em comunidade técnica e como impactar as pessoas por meio da tecnologia.

# PREFÁCIO

Este livro é o resultado de um trabalho voluntário, realizado por 6 amigos MVPs, que têm paixão por compartilhar conhecimento. Quando dizemos voluntário, significa que o ganho que teremos é apenas o aprendizado que cada um de vocês vai adquirir e isto representa para nós o maior “lucro” que teremos.

O objetivo deste livro é ajudar a quem precisa aprender a programar na linguagem Visual C#, uma das mais usadas no mercado mundial. A demanda por profissionais nesta área é enorme, e como não temos desenvolvedores suficientes, incentivamos qualquer pessoa a estudar e ter o conhecimento para entrar na área.

# AGRADECIMENTOS


## ANDRÉ CARLUCCI

Dedico esse livro às minhas 2 filhas, que não se cansam de perguntar o porquê das coisas, à minha esposa que me apoia e me dá energia pra seguir sempre, aos meus amigos e co-autores pela oportunidade, parceria e compreensão durante a escrita e ao leitor, que é o motivo principal de nossa empreitada.

## CARLOS DOS SANTOS

Escrever um livro é um “sonho realizado”, mais do que compartilhar o conhecimento, coisa que já fazemos diariamente em diversos canais, um livro é algo para sempre, e agradeço muito por ter tido esta oportunidade em minha vida.

Agradeço também a todas as pessoas que me ajudaram a chegar até este ponto e muitas delas estão aqui junto comigo neste livro, pessoas que eu admiro e respeito.

A minha família, minha esposa e filhos, que me apoiam e acreditam em todas as ideias, às vezes, meio malucas, que tenho. A todos da CDS Informática, empresa que construímos juntos ao longo de mais de 25 anos.

## CLAUDENIR ANDRADE

Este é meu terceiro livro em 25 anos de trabalho. Impactar vidas e pessoas através da tecnologia é o que me move. Minha esposa é meu apoio, meu braço direito e esquerdo desde o começo desta jornada. Estamos há 20 anos juntos, fiéis a Jeová e um ao outro. Meus filhos são meu porto seguro, e juntos em família vamos construindo nosso futuro. Sou grato a eles por todo apoio em todas as horas.

Pessoas vem antes da tecnologia, pois tecnologias são executadas e construídas por pessoas. Se você não entende de pessoas, você não entende de tecnologia. Este livro tem objetivo de humanizar, de ajudar pessoas, para aqueles que desejam iniciar suas carreiras no desenvolvimento com C#. Sejam bem vindos ao maravilhoso novo mundo da programação, onde o algoritmo te encanta e te embriaga, onde todo esforço e estudo valerá a pena, ainda que no futuro ela se torne obsoleta.

## RAFAEL ALMEIDA

Dedico este livro a minha esposa Thysbe e minhas filhas Eduarda e Heloysa, por todo apoio que tem me dado, escrever um livro era sonho, e agora realizado, dedico a todos que direta ou indiretamente contribuíram para que isso se tornasse possível, um agradecimento especial a todos autores deste livro, pessoas que admiro, por sua dedicação e tempo focando em compartilhar conhecimento para construir um mundo melhor por meio da leitura.

## RAY CARNEIRO

Agradeço a minha esposa Pollyanna e a minha querida filha recém-nascida, Laura, por todo apoio e suporte em todos os projetos que eu me proponho a realizar. Vivemos em um mundo dinâmico onde aprender diariamente é necessário e sem o apoio da minha família nada disso faria sentido.

**RENATO HADDAD**

Este é o meu 14 livro publicado, e a cada um penso o quanto podemos abrir o caminho das pessoas, a fim de prosperar e gerar oportunidades, desafios, propagar o bem a todos através do conhecimento.

Agradeço a todos os amigos e autores deste livro pela dedicação e aceitar o desafio de escrever uma obra. É sempre um trabalho de time, muita sincronia, conhecimento e respeito.

Agradeço a minha esposa e família por proporcionar uma criação de sempre ajudar a quem precisa, sem esperar nada em troca, visto que o universo se encarrega do retorno.

# SUMÁRIO

## ELGIN DEVELOPERS COMMUNITY
## PATROCINADOR DA VERSÃO DIGITAL DESTE LIVRO

A Comunidade do Desenvolvedor de Varejo, Pagamento e Automação Comercial.

Por: Claudenir Andrade

Prezados parceiros e escovadores de Bits! É um prazer, uma honra e um privilégio a Elgin/ Bematech, por meio do programa Elgin Developers Community, patrocinar a versão digital deste livro ao lado de grandes autores.

Por que embarcamos neste sonho, junto com estes autores renomados reconhecidos pela Microsoft? Porque acreditamos em um único propósito: O **conhecimento** transforma e a **comunidade** te prepara para os desafios.

O conhecimento quando compartilhado é satisfatório, impacta pessoas, transforma vidas e realiza sonhos. É nisso que acreditamos, neste propósito transformador de **digitalizar o varejo brasileiro, o sistema de pagamentos e a automação comercial**, com esta missão de inovar, de transformar.

Isso não se faz sozinho. Isso se faz com dois pilares, **Conhecimento e Comunidade.** O Conhecimento transforma e a comunidade te ajuda a vencer os desafios. Ao longo destes 25 anos pude pessoalmente criar várias comunidades, a maior delas com quase 7.000 ISVs (Software Houses) de varejo. Se você é desenvolvedor, está iniciando e ainda não faz parte de uma comunidade, busque uma que você se identifica, que fala sua língua, a tecnologia que você está em busca de conhecimento. A Microsoft foi a primeira empresa de tecnologia mundial a apostar em comunidades, desde 1997 com o lançamento visual Studio (que ainda não era .NET) no

Developers Days e PDC – Professional Developers Conference no Brasil.

A Comunidade técnica te aproxima de pessoas com o mesmo objetivo, com os mesmos problemas e desafios. Juntos, através deste networking, você pode superar os desafios e crescer como profissional.

Na Elgin/Bematech criamos o **Elgin Developers Community** exatamente com este objetivo, de ajudar os desenvolvedores a superar os desafios do mercado de automação comercial. Com o objetivo de capacitar os desenvolvedores em produtos inovadores e entregar um framework de integração que acelera seu desenvolvimento com impressora, sat, nfce, Android, pagamentos. Nosso propósito com o Elgin Developers Community é sermos seu provedor de tecnologia para a automação comercial e pagamentos, rodando Windows, Linux e Android.

O Mundo mudou, está mais aberto, menos competitivo em termos de divisão tecnológica, hoje a palavra do momento é interoperabilidade. Um pouco antes de finalizar este livro, visitei Luciano Palma, outro "monstro" da comunidade de desenvolvedores no Brasil, com passagem pela Microsoft e Intel. Ele estava me apresentando nosso PDV Android Elgin/Bematech M10., rodando com o sistema dele, o PEDSeguro. Nesta ocasião ele estava usando um MACBookPro, rodando Sistema operacional da Apple, Codando com Visual Studio Code da Microsoft, "Buildando" um app para Android. Consegue imaginar isso há dez anos atras? Apple, Microsoft e Google em um único projeto de Varejo, de automação comercial? O que parecia impossível, hoje é realidade através da interoperabilidade.

Este é o cenário que nos encanta aqui na Elgin/Bematech, a multidisciplinaridade em tecnologia. Estamos apostando forte em Windows e Android para automação comercial, já somos protagonistas neste mercado, **hoje a Elgin/Bematech é o maior e único fabricante** a ofertar code-sample em 12 linguagens Android para Automação comercial https://github.com/elgindevelopercommunity

Até o lançamento deste livro registramos 2.500 ISVs (Software Houses) em nossa comunidade. Quer fazer parte dela?

https://www.elgin.com.br/Automacao clique no link “Elgin Developers” e participe, faça parte da maior comunidade de desenvolvedores de Varejo, Automação Comercial e Pagamentos do Brasil e esteja na frente de todas as inovações conosco.

A missão de **Digitalizar o Varejo** não se faz sozinho. Acreditamos que só conseguiremos cumprir esta missão através de vocês: Software houses, Software houses/Revendas e Revendas que representam uma software houses. Será com vocês, através de vocês e junto com vocês que cumpriremos esta missão maior e este propósito que está em nossas mãos executar.

Faça parte do Elgin Developers Community

Claudenir Andrade – claudenir.andrade@elgin.com.br
Diretor de Integração Tecnológica e Marketing da Elgin/Bematech
Regional Director Microsoft / MVP Microsoft

# CAPÍTULO 1 – INTRODUÇÃO

Vivemos em uma era fantástica onde o uso da tecnologia não está presente apenas em placas de computadores e celulares inteligentes mas também em aparelhos domésticos, carros elétricos e qualquer coisa que consiga se conectar a uma rede de computadores.

Todas essas facilidades e benefícios do mundo moderno dependem de serviços inteligentes que funcionam em computadores, sejam eles portáteis ou não. O grande maestro dessas inteligências e facilidades se chama software.

Softwares são programas de computadores, aplicativos de celulares ou até mesmo instruções de inteligência artificial que auxiliam tarefas repetitivas ou até mesmo tarefas preditivas a serem executadas e para isso é preciso que essas rotinas sejam previamente programadas por um ser humano e previamente compiladas por um computador para que o resultado final seja um software.

O mercado de software mundial evolui em grande escala a cada dia que passa e encontrar profissionais que entendam desse mundo tecnológico se tornou um grande desafio. Esses profissionais que transformam os desafios do mundo real em tecnologia são as famosas pessoas programadoras de computadores.

A profissão de pessoa programadora de computadores cresceu exponencialmente nos últimos anos e tende a crescer cada vez mais. Em todo o mundo cada vez mais investe-se em pesquisas e

desenvolvimentos de produtos e processos e o papel de pessoas programadoras se tornou essencial para o crescimento de uma empresa, afinal, toda empresa do mundo, direta ou indiretamente precisa de software.

Quando se inicia um novo projeto de software, normalmente o objetivo é criar soluções inovativas que aceleram ou automatizam tarefas cotidianas. As soluções podem ser diversas e atenderem a diversos tipos de mercado. Por exemplo, para o mercado de entretenimentos temos diversas opções de sistemas ou aplicativos, tais como Instagram, Facebook e demais redes. Talvez você não saiba, mas a tecnologia e a programação são os pilares dessas empresas e não apenas entregam entretenimento mas também geram novas formas de comunicação, novas formas de trabalhar e novas formas de ver o mundo.

As gerações mais novas já nasceram com todas essas facilidades na palma da mão, mas antigamente a coisa não era exatamente desse jeito. Os desafios tecnológicos eram grandes devido a falta de capacidade de processamento de informações, que era limitada. Hoje em dia não temos esse problema e os recursos são tão grandes quanto nossa imaginação!

O primeiro passo para ingressar no mundo de desenvolvimento de software é escolher uma plataforma de desenvolvimento e uma linguagem de desenvolvimento. Para isso, não existe nenhuma fórmula secreta e nem segredos escondidos a sete chaves, basta apenas pesquisar sobre o mercado de software, sobre as tendências de plataformas e linguagens e escolher qual estudar.

Uma linguagem robusta, com anos de suporte e adotada largamente pelo mercado de desenvolvimento de software profissional é o C# (pronuncia-se “C Sharp”). O C# é uma linguagem de programação moderna, robusta e orientada a objetos que está em constante evolução e adoção de profissionais no mundo inteiro. Com essa linguagem de programação é possível criar softwares para muitos tipos de aplicativos tais como celulares, computadores pes-

soais, jogos e muito mais!

O C# tem raízes na família de linguagens C e os programadores que já conhecem C ou C++ terão facilidade em migrar ou se atualizar para esse novo paradigma de programação. Por outro lado, para aqueles que nunca tiveram contato com a linguagem, a adoção e curva de aprendizado também é facilitada pela enorme comunidade técnica que a utiliza, pelos diversos sites e cursos que estão disponíveis.

Originalmente a linguagem foi criada pela Microsoft e tinha como objetivo fornecer alternativa ao mercado de software para criação de novos sistemas voltados para a plataforma Windows, porém, ao longo do tempo a linguagem foi evoluindo e conquistando mais adeptos não apenas do mercado corporativo, mas também do mundo do código-aberto. Atualmente é possível criar sistemas que rodam em Windows, Linux e Mac com a mesma linguagem de programação.

Isso é um trunfo enorme pois antigamente normalmente era necessário criar um sistema específico para cada tipo de plataforma, aumentando o custo e o tempo de entrega do projeto, sem contar a necessidade de encontrar um especialista para cada tipo de plataforma. Esse problema foi sanado com o suporte das linguagens e também das plataformas de se adaptarem a multiplataformas.

Como já vimos anteriormente, o C# é uma linguagem de programação orientada a objetos e orientada a componentes. Isso significa que a linguagem suporta construções e implementações de forma nativa para que nós como pessoas programadoras consigamos trabalhar nos projetos de uma forma clara e objetiva com recursos que são implementados na própria linguagem.

Por exemplo, se precisarmos construir um sistema que fará leituras de arquivos no formato XML, a própria linguagem de programação já nos provê implementações para que consigamos ler e manipular esse tipo de arquivo de forma fácil e intuitiva sem ter

que implementar lógicas de programação para executar a mesma tarefa.

Desde a sua criação o C# adiciona com frequência recursos para dar suporte a novas cargas de trabalho e práticas de design de software emergente. Com o C# podemos criar softwares nativamente para rodar em sistemas em nuvem, aplicativos para dispositivos móveis, sistemas embarcados, websites, aplicações para web, portais e muito mais. Por se tratar de uma linguagem de programação com forte adoção e em constante desenvolvimento, as opções são infinitas.

Para conseguirmos trabalhar com essa linguagem de programação precisamos primeiro entender o ecossistema necessário para iniciar os projetos. A forma como utilizamos a linguagem C# para criar sistemas é através de uma IDE. Uma IDE é um ambiente de desenvolvimento integrado. Na prática, é um software que precisaremos baixar e instalar e ele vai nos possibilitar selecionar qual tipo de projeto queremos criar, qual linguagem de programação usaremos e quais configurações e extensões queremos obter.

Atualmente existem diversas IDEs no mercado que atendem essa necessidade, porém as mais famosas e mais relevantes são de fato o Visual Studio e o Visual Studio Code. Essas IDEs foram criadas pela Microsoft com objetivo de facilitar o nosso desenvolvimento de sistemas e com essas ferramentas conseguimos criar, analisar e distribuir softwares.

O Visual Studio é uma IDE consolidada há anos e também é evoluída com frequência pelo time de Engenheiros da sua fornecedora, sendo uma ótima opção para quem deseja iniciar com desenvolvimento de software. Visando atingir um público abrangente, é possível instalarmos e utilizarmos o Visual Studio no Windows, Linux ou Mac.

Existem diversos tipos de licenciamentos que precisam ser considerados para seu projeto, caso o projeto seja corporativo e haja

um time grande de pessoas desenvolvedoras, será necessário obter a licença adequada do produto, porém, para fins educativos ou não comerciais existe a versão da comunidade, sem custo.

Para iniciar, basta visitarmos o site do Visual Studio (https://visualstudio.microsoft.com) e escolher a versão para o sistema operacional aderente. Nesse livro usaremos a versão Visual Studio Code.

O Visual Studio Code não é uma IDE, ele é um editor de código-fonte desenvolvido pela Microsoft para Windows, Linux e MacOS. Apesar de não ser uma IDE completa, ele inclui suporte para depuração de códigos, controle de versionamento de código com GIT incorporado, complementação de código inteligente e muito mais. O interessante dessa versão do Visual Studio é que você pode instalar diversas extensões diferentes para auxiliar no seu dia a dia. Outro ponto interessante é que você pode editar e utilizar diversas linguagens de programação no mesmo editor.

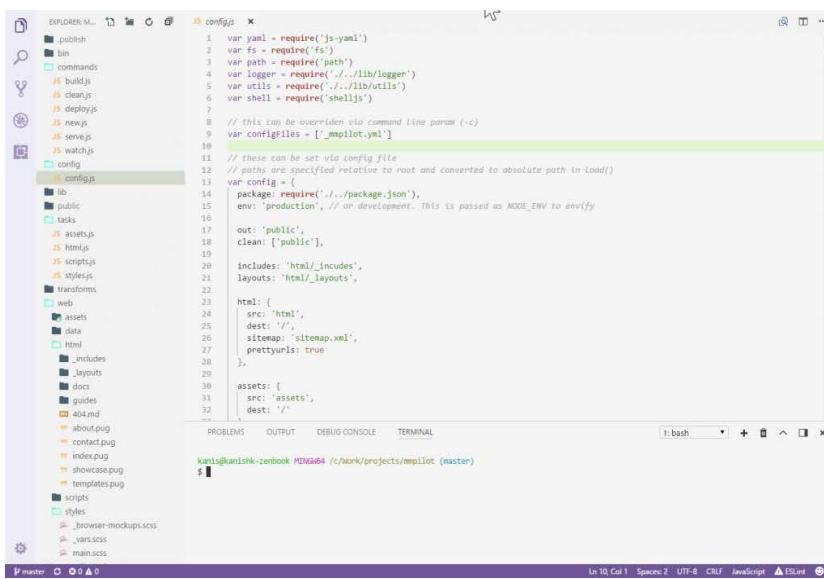



Figura 1 - Visual Studio Code

Essas extensões são uma boa opção pois você pode maximizar seu tempo e efetuar tarefas customizadas através das extensões ao invés de precisar fazer a mesma tarefa em outro local. Por exemplo, com o Visual Studio Code é possível instalar extensões para gerenciar recursos que estão criados dentro da nuvem do Microsoft Azure e a partir da linha de comando dentro do editor de código é possível se conectar à nuvem e executar comandos remotamente.

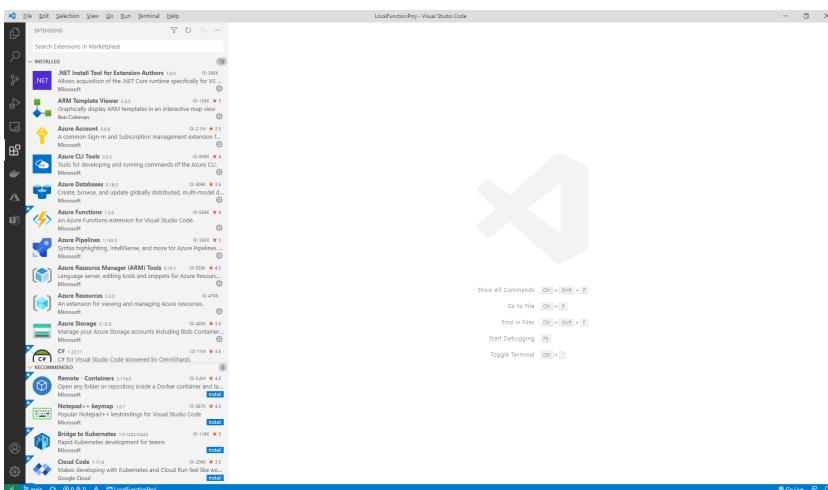



Figura 2 - Visual Studio Code com extensões

Além das linguagens de programação para criar softwares, precisaremos utilizar um framework. Um framework é uma abstração de funcionalidades que são criadas e distribuídas para que as pessoas programadoras possam instalá-lo e usufruir das funcionalidades e do ambiente de compilação que ele propicia.

O .NET Framework é uma iniciativa da empresa Microsoft, que visa criar uma única plataforma de desenvolvimento e execução de sistemas e aplicações. Dessa forma, todo e qualquer código gerado em .NET pode ser executado em qualquer dispositivo que possua o framework instalado.

O .NET Framework consiste basicamente em dois componentes

principais, sendo executada sobre uma **Common Language Runtime – CLR**, um ambiente de execução independente de linguagem.

O trabalho da CLR é executar diversas linguagens, tais como C#, F# e VB.NET e permitir grande interoperabilidade entre elas. Além disso, ele fornece gerenciamento de memória, controle de exceção, manipulação de processamento paralelo e concorrente, reflexão, segurança, serviços de compilação para a arquitetura específica, entre outros.

Originalmente só funcionava no Windows, porém ao longo do tempo e com a grande adoção de sistemas operacionais livres o .NET Framework adaptou-se ao mundo open-source. O .NET Core já nasceu como um projeto 100% aberto, contando com contribuições da comunidade mundial através da .NET Foundation.

Outro componente de grande importância para ser citado é o CIL (Common Intermediate Language). Ela é uma linguagem de programação de baixo nível do ambiente de programação da Microsoft. O código de mais alto nível do ambiente .NET Framework, por exemplo o C#, é compilado em código CIL, que é assemblado em código chamado bytecode.

CIL é um código orientado a objeto e executado por uma máquina virtual. A CIL tinha inicialmente o nome de Microsoft Intermediate Language (ou MSIL), na época das versões beta da linguagem .NET. Depois da estandardização do C# e da CLI, o bytecode foi oficialmente referenciado sob a designação de CIL.

A figura a seguir representa os conceitos apresentados acima, onde no topo temos um código fonte em alguma linguagem de programação que são convertidos para linguagem de baixo nível, ou linguagem de máquina.

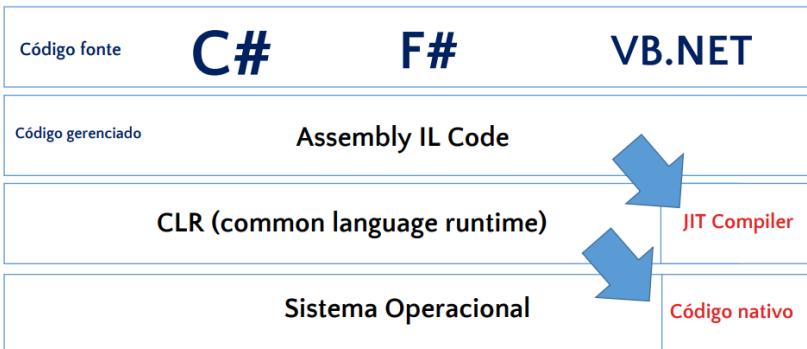



Figura 3 – CLR do .NET

Após todo processo de compilação dos códigos de alto e baixo nível, a execução de comandos com o sistema operacional é intermediada pelo framework instalado e assim o software executa suas tarefas. Vale ressaltar também que o processo de compilação do código de alto nível para baixo nível resultará em uma ou mais DLLs (dynamic-link library). DLL se trata de uma biblioteca dinâmica que contém os dados que podem ser acessados pelo seu programa no computador e também por outros programas, inclusive de forma simultânea.

Dessa forma, toda vez que compilamos nossos softwares com o .NET framework, ele se encarregará de fazer a geração das DLLs do nosso sistema. Uma DLL pode conter códigos, dados e outros recursos computacionais e existem diversos tipos de DLLs para diversos tipos de funções. O próprio .NET Framework fornece um conjunto de DLLs pré-existentes que podemos agregar aos nossos projetos quando necessário.

A vantagem da utilização das DLLs se justifica na economia de recursos utilizados pelo sistema operacional. Quando vários programas usam a mesma biblioteca de funções, por exemplo, o uso de memória física pode ser reduzido, melhorando o desempenho dos sistemas rodando.

Além do benefício do desempenho, as DLLs também são uma estratégia para organizar os softwares de forma modular, facilitando o desenvolvimento de aplicações. Além disso, quando uma DLL necessita de manutenção, não é necessário reinstalar todo o sistema, bastaria apenas alterar a DLL que foi alterada.

# CAPÍTULO 2 – CONCEITOS BÁSICOS DA LINGUAGEM C#

Neste capítulo, analisaremos os conceitos básicos da estrutura da linguagem C#. É muito importante que saibamos como a linguagem se comporta não apenas com suas funções, mas também com as melhores práticas de gerenciamento do projeto de software que você esteja planejando ou atualmente trabalhando.

## NAMESPACES

Os namespaces são usados no C# para organizar e prover um nível de separação de código fonte. Podemos considerá-lo como um "container" que consiste de outros namespaces, classes, etc. Normalmente um namespace pode conter os seguintes membros:

- Classes
- Interfaces
- Estruturas
- Delegates

Vamos entender o conceito de namespaces em um cenário um pouco mais real. Imagine que temos um grande número de arquivos e pastas em nosso computador. Imagine então como seria difícil gerenciar esses arquivos e pastas que foram colocados no mesmo diretório. Por esse motivo seria muito melhor se dividíssemos os arquivos em pastas diferentes onde faz sentido eles estarem, correto? A mesma analogia pode ser aplicada a utilização de namespaces.

No C# a utilização de namespaces é similar ao exemplo dado anteriormente. O namespace nos ajuda a organizar diferentes tipos de membros que estão relacionados no mesmo lugar. Uma ou mais classe podem conter o mesmo nome, porém devem estar em namespaces diferentes. Se criarmos duas classes com o mesmo nome dentro do mesmo projeto, por exemplo, teremos um erro de compilação pois teremos um conflito de nomeação. Veja um exemplo.

```
Namespaces.cs 1, U
Cap2 > Namespaces.cs > {} ProgramPrincipal > ProgramPrincipal.ClasseUm
1   namespace ProgramPrincipal{
        0 references
2       public class ClasseUm
3       {
            0 references
4           public int Id { get; set; }
            0 references
5           public st
6       }
7
8       public class ClasseUm
9       {
10          public int Id { get; set; }
11          public string Nome { get; set; }
12      }
13
14  }

class ProgramPrincipal.ClasseUm
The namespace 'ProgramPrincipal' already contains a definition for 'ClasseUm'
[livrocsharp] csharp(CS0101)
View Problem (Alt+F8)   No quick fixes available
```

Figura 4 – Namespaces com classes contendo nomes iguais

Na figura acima temos duas classes com o nome de “ClasseUm”. Isso não é permitido na estrutura da linguagem e o compilador vai exibir um erro informando que já existe uma classe com o mesmo nome. Assim como em qualquer sistema operacional, dentro de uma mesma estrutura não será possível ter um arquivo com o mesmo nome.

Assim que alterarmos o nome da segunda classe para algo diferente, o compilador entenderá que não temos mais conflitos de membros dentro do mesmo namespace.

```
Namespaces.cs U ×
Cap2 > Namespaces.cs > {} ProgramPrincipal
1   namespace ProgramPrincipal{
2       public class ClasseUm
3       {
4           public int Id { get; set; }
5           public string Nome { get; set; }
6       }
7
8       public class ClasseDois
9       {
10          public int Id { get; set; }
11          public string Nome { get; set; }
12      }
13
14  }
```

Figura 5 – Namespaces com classes contendo nomes diferentes

## ACESSANDO OS MEMBROS DE UM NAMESPACE

Os membros de um namespace podem ser acessados através de um operador de ponto (.). A sintaxe para acessar o membro dentro do namespace é **NomeDoNameSpace.NomeDoMembro**. Por exemplo, se temos uma classe chamada **ClasseUm** e queremos acessá-la, podemos fazer da seguinte maneira:

```
AcessandoNameSpaces.cs U ×
Cap2 > AcessandoNameSpaces.cs > {} MyProgram
1   namespace MyProgram
2   {
3       public class ClasseUm
4       {
5           public static void Main()
6           {
7               ProgramaPrincipal.ClasseDois ClasseDois = new ProgramaPrincipal.ClasseDois();
8           }
9       }
10  }
```

Figura 6 - Acessando um objeto de outro namespace

Perceba que na figura anterior estamos desenvolvendo nosso código dentro do namespace **MyProgram**. Esse namespace não tem visibilidade do namespace ProgramaPrincipal, dessa forma, podemos indicar no programa qual é a localidade dele.

Apesar dessa forma funcionar, existe uma forma mais amigável e prática de obtermos a visibilidade do outro namespace e, para isso podemos usar a palavra-chave using.

```
AcessandoNameSpaces.cs U ×
Cap2 > AcessandoNameSpaces.cs > {} MyProgram
1   using ProgramaPrincipal;
2
3   namespace MyProgram
4   {
        0 references
5       public class ClasseUm
6       {
            0 references
7           public static void Main()
8           {
9               ClasseDois ClasseDois = new ClasseDois();
10          }
11      }
12  }
```

Figura 7 - Utilizando a palavra-chave using

Veja na figura anterior que removemos o nome do namespace antes da classe **ClasseDois** e adicionamos no topo do programa a palavra-chave using, indicando onde ir buscar a referência do objeto. Não existe um padrão ou melhor forma de utilizar, ambos funcionam, porém, é muito comum ver no mercado a utilização do using no topo do programa.

A principal funcionalidade do namespace é de fato organizar o projeto. A medida que ele vai ficando maior e com mais arquivos é extremamente importante que saibamos como segregar o projeto visando sobre a responsabilidade de cada componente e determinando suas ações de forma isolada.

Uma boa prática recomendada pela Microsoft e diversas pessoas programadoras profissionais é criar a estrutura do seu projeto seguindo a seguinte sintaxe:

*NomeDaEmpresa.NomeDoProjeto.CamadaDoProjeto.Funcionalidade*

Exemplo:

Microsoft.LivroCSharp.CamadaDeDados.ConectorSqlServer

## CLASSES

Antes de falarmos tecnicamente sobre o que é uma classe, vamos primeiro fazer algumas analogias para entender o porquê precisamos de classes e objetos no mundo de desenvolvimento de softwares. Nos próximos capítulos veremos mais detalhes sobre programação e objetos para complementar nosso aprendizado.

Vamos imaginar que nosso primeiro projeto de software seja a criação de um sistema para uma agência bancária. Nessa agência bancária temos diversos atores e processos que compõem o negócio. Quando falamos em atores, imagine a funcionária do caixa, o gerente da agência e todos os funcionários ou processo que fazem com que aquele negócio funcione.

Todo dia de manhã ao abrir a agência, o gerente precisa abrir os caixas para que os funcionários que são operadores do caixa possam trabalhar e efetuar suas rotinas diárias de crédito e débito em contas correntes.

Veja que apenas em uma breve narração conseguimos identificar dois atores (gerente e funcionários do caixa) e dois processos de negócio (crédito e débito em conta bancária).

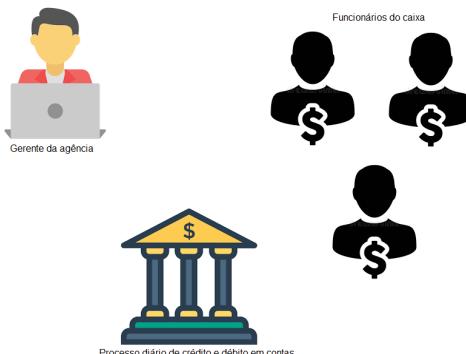



Figura 8 - Representação do domínio de negócio

A figura anterior representa o processo que denominamos como análise de domínio de negócio. Faz parte identificar nesse processo os principais atores e quais processos são executados por eles, afinal, queremos construir um sistema para auxiliar ou melhorar esses processos, não é mesmo? Essa análise efetuada nos ajudou a saber quem é quem e quem faz o que.

Agora, precisamos representar isso no sistema e utilizar as classes é a forma como faremos isso! A classe é uma abstração do mundo real em programação. Ela vai modelar no sistema quais atores ou processos existirão e como eles se comportam. Esses atores possuem características comuns tais como nome, idade e certamente um número de CPF, dessa forma, a classe também precisará refletir essas propriedades.

Criaremos uma classe no Visual Studio Code chamada Gerente.cs e colocaremos o seguinte código:

```
namespace SistemaBancario {
    public class Gerente {
        public string Nome { get; set; }
        public int Idade { get; set; }
        public int Cpf { get; set; }
    }
}
```

Basicamente o que fizemos aqui foi identificar quais atributos o gerente possui e são relevantes para o sistema e quais propriedades criaríamos na classe para que fosse utilizado em nosso sistema.

```
namespace SistemaBancario {

    public class Gerente {

        public string Nome { get; set; }

        public int Idade { get; set; }

        public int Cpf { get; set; }

    }

}
```

Figura 9 - Representação entre o ator Gerente e a classe Gerente.cs

Até o momento temos apenas as propriedades da classe que podem ser utilizadas em nosso sistema e nada mais. Porém, no papel de Gerente esse ator certamente efetua tarefas no seu dia a dia tais como aprovar horas de funcionários, criar novas contas de clientes e diversas outras atividades.

Essas atividades são refletidas em nossa classe através de **métodos**. Os métodos representam computacionalmente uma ação feita pelo ator na vida real. Dessa forma, vamos alterar um pouco nosso código e incluir um método que vai possibilitar o gerente aprovar as horas de um funcionário informando o cpf do funcionário.

```
namespace SistemaBancario{
    public class Gerente {
        public string Nome { get; set; }
        public int Idade { get; set; }
        public int Cpf { get; set; }
       public void AprovarHorasDeFuncionarios(int cpf){
           //lógica para aprovar horas
        }
    }
}
```

A estrutura do código anterior representa uma classe no C#,

uma representação do mundo real descrita através de um código fonte onde o computador possa entender e executar as tarefas de forma automatizada. Esse processo feito até agora representa a modelagem de domínio do negócio e normalmente em um projeto real é feito junto com especialistas que entendem do negócio em questão.

Idealmente é uma boa prática explorar os especialistas de negócio, criar fluxos e mapeamentos do trabalho que é executado no ambiente que necessita de um sistema e posteriormente modelar o sistema. Nos próximos capítulos entraremos em maiores detalhes sobre programação orientada a objetos para massificar nosso entendimento acerca do assunto.

## OBJETOS

Conforme avaliamos anteriormente as classes são abstrações da vida real, ou seja, elas descrevem de uma forma computacional quais os atores e processos existentes no mundo real.

Até aqui sabemos que as classes têm propriedades e métodos, porém não sabemos como o computador ou compilador da linguagem vai entender e processar essas informações. É aqui que os objetos têm um papel fundamental.

O objeto nada mais é que a compilação deste código escrito por nós sendo colocado na memória do computador para que ele seja interpretado e executado.

Esse processo de criar o objeto e colocá-lo na memória é denominado instanciamento de classe. Para isso, basta criar um código similar ao demonstrado abaixo:

```
namespace SistemaBancario{
    public class InstanciaObjeto
    {
        public static void Main()
        {
            var objetoGerente = new Gerente();
        }
    }
}
```

A utilização da palavra reservada new no C# é responsável por analisar o código feito na classe, e então, disponibilizar na memória do computador para utilização.

Em tempo de desenvolvimento após criar a instância da classe, é possível analisar quais atributos e métodos à classe possui apenas colocando um ponto (.) após o nome da variável, conforme figura a seguir:

```
InstanciaObjeto.cs 2, U
Cap2 > InstanciaObjeto.cs > {} SistemaBancario > SistemaBancario.InstanciaObjeto > Main()
1   namespace SistemaBancario{
        0 references
2       public class InstanciaObjeto
3       {
            0 references
4           public static void Main()
5           {
6               var objetoGerente = new Gerente();
7               objetoGerente.
8           }       AprovarHorasDeFuncionarios
9       }           Cpf
10  }               Equals
                    GetHashCode
                    GetType
                    Idade
                    Nome
                    ToString
```

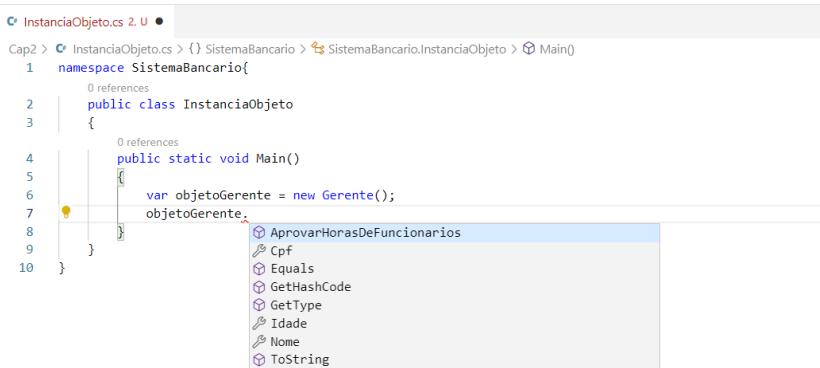



Figura 10 - Instância de classe, propriedades e métodos

O Visual Studio Code detém uma funcionalidade chamada Intellisense, ela é responsável por analisar em tempo de desenvolvimento quais propriedades e métodos existem na classe e demonstrar isso visualmente.

## VARIÁVEIS

As variáveis são espaços na memória do computador onde podemos salvar, manipular e exibir informações. Vamos analisar novamente o código da imagem anterior.

```
namespace SistemaBancario{
   public class InstanciaObjeto
   {
      public static void Main()
      {
         var objetoGerente = new Gerente();
      }
   }
}
```

A palavra reservada **var** tem a função de informar o sistema que ali está ocorrendo a criação de uma variável. Nesse exemplo, estamos criando um novo objeto e atribuindo seu valor e estado na variável objetoGerente. É possível também criar a variável com o tipo dela implicitamente ao invés de usar a palavra **var**.

As variáveis têm uma importância enorme no mundo de desenvolvimento de software pois são nelas que guardamos, manipulamos, alteramos e exibimos valores para os usuários finais ou simplesmente fazemos cálculos e manipulações e enviamos talvez para outros sistemas que queiram se comunicar.

De forma objetiva, a variável é meramente um nome dado para um armazenamento de dados que nosso programa vai manipular. Cada variável no C# possui um tipo específico que determina seu tamanho e o quanto de informação ela pode salvar na memória.

Os tipos básicos são categorizados em:

| Tipo | Exemplo |
|---|---|
| Tipos integrais | sbyte, byte, short, ushort, int, uint, long, ulong, and char |
| Tipos de ponto flutuante | float and double |
| Tipos decimais | decimal |
| Tipos booleanos | true or false values, as assigned |
| Tipos nulos | Nullable data types |

Existem outros tipos de variáveis também, como por exemplo *Enums* e tipos por referência como classes. A forma que definimos variáveis no C# é:

**<tipo de dados> <nome da variavel> = <valor>;**

Por exemplo:

```
namespace SistemaBancario{
   public class Variaveis
   {
      public static void Main()
      {
         int cpf = 123456789;
         string nome = "Ray Carneiro";
         decimal salario = 1000;
         bool funcionarioAtivo = true;
      }
   }
}
```

No código anterior temos alguns exemplos de variáveis e tipos diferentes. Visualizando o nome das variáveis podemos perceber que cada uma delas tem uma finalidade, um tipo e um valor inicial. Assim que essas variáveis ficam disponíveis na memória é possível

acessá-las, exibi-las ou manipulá-las. De fato, as variáveis são formas como podemos armazenar as informações durante o fluxo a ser feito dentro do sistema.

# CAPÍTULO 3 – FUNÇÕES INTERNAS DO C#

O C# contém diversos métodos internos da linguagem que permitem manipular textos, datas, fazer contas com números, e isto usamos em diversas ocasiões no dia a dia. Tais funcionalidades existem para facilitar a vido do desenvolvedor, senão seria preciso construir uma lógica, ocasionando possibilidades de erros.

Neste capítulo iremos abordar as mais usadas de forma simples e prática. Para isto, no VS Code, crie uma nova pasta chamada Cap3 onde teremos todos os exercícios C#. Você pode clicar no segundo ícone, conforme a figura a seguir ou clicar com o botão direito na lista de itens da janela do Explorer e selecionar New Folder.

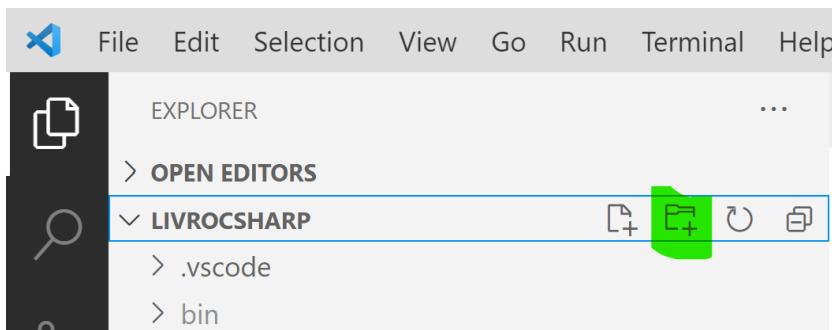



Figura 11 - Nova pasta CAP3

Sendo assim, todos os exercícios deste capítulo serão criados dentro desta pasta Cap3, portanto, para adicionar um novo arquivo esta pasta tem que estar selecionada. Clique no primeiro ícone (New File), conforme a figura a seguir, ou com o botão direito sobre

a pasta e selecione New File.

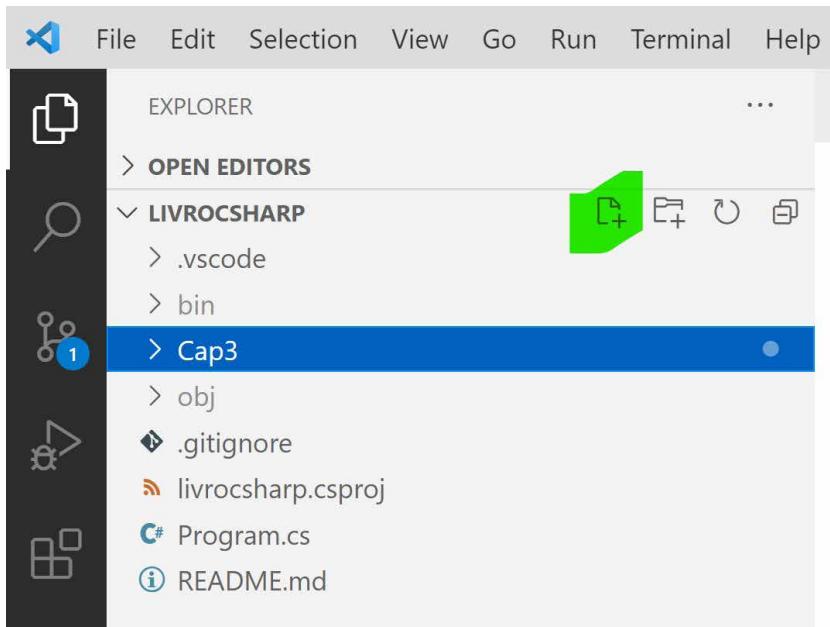



Figura 12 – Novo arquivo

## FUNÇÕES DE TEXTOS

As funções de textos, como o próprio nome diz, são usadas para se manipular *strings*, cadeias de textos, pois nem sempre teremos o conteúdo limpo ou do jeito que precisamos. Com isto, podemos fazer o que quiser, limpar espaços em branco, obter a quantidade de caracteres, transformar tudo para maiúscula ou minúscula, trocar o conteúdo, enfim, depende do contexto.

Vamos adicionar o arquivo funcoesTexto.cs. Não se esqueça da extensão .cs para que o compilador saiba que se trata de um arquivo C#. Em seguida, digite a lista de USING, o namespace, o nome da classe e o SVM (static void Main).

```
using static System.Console;
```

```
using System;
using System.Linq;

namespace livrocsharp
{
    class funcoesTexto
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Como estamos usando uma aplicação de Console, a interação ocorre em mostrar o resultado na tela através do comando *Console.WriteLine*(expressão). No entanto, o código fica muito grande, e para isto podemos na lista de *using* referenciar o *namespace* deste comando de forma estática, a fim de apenas usar o método *WriteLine*(expressão). Veja que isto está explícito na primeira linha (*using static System.Console;*). Tudo que estiver dentro deste *namespace* pode ser referenciado apenas o método. Em seguida, temos os outros *namespaces* que usaremos no código.

Tudo o que for digitado para ser executado deverá estar contido dentro do SVM (static void Main).

## TRIM

Este método retira todos os espaços em branco antes e depois de uma expressão. Caso tiver espaços em branco no meio da expressão, teremos que construir um código para retirar, não é o caso do TRIM. Este método é usado em interações com usuários, normalmente em cadastros onde são digitadas muitas informações em uma tela. Ou ainda, em casos de trazer informações de outra fonte de dados, arquivos textos, CSV, Excel, dados exportados de bancos de dados, enfim, o TRIM nos ajuda muito na consistência de capturar apenas a expressão em si.

Digite os seguintes comandos no SVM, o WriteLine exibirá o texto “------ Funções de Textos -------”. Em seguida criamos uma variável chamada empresa contendo um texto com espaços em branco antes e após os textos, justamente para simularmos o uso do TRIM. Em seguida, temos um texto a ser exibido ao usuário para saber o resultado.

O comando ***WriteLine*****($”texto {comando}”)** é o que chamamos de *Interpolação de Strings*, onde a sintaxe do C# permite colocar tudo dentro de uma expressão “...”, sejam textos ou comandos. No entanto, é **obrigatório** iniciar com o caractere **$ (cifrão)**. E, todos os comandos (variáveis e comandos) deverão estar entre chaves **{ ... }**.

```
static void Main(string[] args)
{
    WriteLine("------ Funções de Textos -------");
    string empresa = " Microsoft Corporation   ";

    WriteLine("TRIM - retira os espaços em branco 
antes e após a expressão");
   WriteLine($"Nome sem espaços: {empresa.Trim()}");
}
```

Neste caso, note que o comando a seguir usamos um texto “Nome sem espaços: ”, seguido, entre chaves, nome da variável *empresa + método TRIM().* Basta respeitar a sintaxe que o compilador entenderá perfeitamente. Esta sintaxe tem a vantagem de deixar o código mais limpo e de fácil entendimento, caso contrário, teria que ficar concatenando expressões e a sintaxe ficaria enorme.

WriteLine($”Nome sem espaços: {empresa.Trim()}”);

Com o código pronto, podemos executar o projeto. Como o C# permite ter apenas um SVM *(static void Main)* por projeto, abra o arquivo **livrocsharp.csproj** e adicione a linha do <*StartupObject*>

contendo o nome do namespace + classe a ser executada, neste caso, **livrocsharp.funcoesTexto**. Ao final, salve o arquivo CTRL + S.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
          <StartupObject>livrocsharp.funcoesTexto</StartupObject>
  </PropertyGroup>
</Project>
```

Agora sim, o VS Code já sabe qual classe será executada. No menu *Run*, selecione *Start Debbuging* ou pressione apenas F5, conforme a figura a seguir.

**Figura 13 – Executar o código**

No VS Code será aberta a janela DEBUG CONSOLE para mostrar o resultado da execução. Para efeito do livro vou omitir a lista de DLLs listadas nesta janela e mostrar apenas o resultado, conforme a seguir. No resultado temos todo o conteúdo da variável empresa sem os espaços em branco no início e fim.

***Resultado:***

**—— Funções de Textos ——**

```
TRIM - retira os espaços em branco antes e após a expressão
    Nome sem espaços: Microsoft Corporation
```

## LENGTH

O método *Length* conta a quantidade de caracteres da expressão, incluindo os espaços em branco em toda a expressão. É muito usado quando precisamos definir um tamanho limite que caiba num campo ou espaço a ser impresso ou mostrado. Em aplicativos onde a interação com usuário é feita, usamos para checar a quantidade de caracteres digitados, em importação de dados de outras fontes, checamos se o tamanho da expressão está de acordo com o necessário, e caso seja preciso podemos até descartar o restante, por exemplo, em um endereço completo, aplicamos uma regra de no máximo 50 caracteres. Caso a expressão tenha 60, 70 caracteres, por exemplo, pegamos apenas os 50 primeiros. Enfim, há diversos casos de uso do *Length*.

Para testes, adicione o bloco de linhas abaixo do Length, o qual é mostrada a quantidade de caracteres da variável empresa (empresa.Length), antes e após o uso do TRIM. Note que a própria variável empresa foi atribuído o comando empresa.Trim() para retirar os espaços em branco, deixando-a limpa como deve ser usada.

```
static void Main(string[] args)
{
    WriteLine("------ Funções de Textos -------");
    string empresa = " Microsoft Corporation   ";
    WriteLine("TRIM - retira os espaços em branco
antes e após a expressão");
   WriteLine($"Nome sem espaços: {empresa.Trim()}");

    WriteLine("Length - retorna a qtde de caracteres");
   WriteLine($"Tamanho do texto: {empresa.Length}");
```

```
    empresa = empresa.Trim();
     WriteLine($"Tamanho do texto após o TRIM(): {empresa.Length}");
  }
```

Salve o código e pressione F5 para executar o código e veja o resultado. Antes a expressão mostrou o tamanho de 25 caracteres e após o TRIM 21. Com isto, já podemos saber com exatidão como contar o tamanho de uma expressão.

***Resultado:***

**—— Funções de Textos ——**

**TRIM - retira os espaços em branco antes e após a expressão**
**Nome sem espaços: Microsoft Corporation**
**Length - retorna a qtde de caracteres**
**Tamanho do texto: 25**
**Tamanho do texto após o TRIM(): 21**

## TOUPPER

O método *ToUpper* tem a função de transformar toda a expressão em letras maiúsculas. É usado em casos de formatação, onde precisamos chamar a atenção como nome, cargo, informações críticas, etc. O uso é simples, basta informar a cadeia de caracteres ou variável, seguido do método *ToUpper()*.

No nosso código, ocultarei algumas linhas dos códigos anteriores para não ficar repetitivo. Digite as linhas para mostrar a variável empresa em letras maiúsculas, sendo *empresa.ToUpper()*.

```
namespace livrocsharp
{
  class funcoesTexto
  {
```

```
        static void Main(string[] args)
        {
           WriteLine("------ Funções de Textos -------");
           ...
            WriteLine($"Tamanho do texto após o TRIM()
: {empresa.Length}");

           WriteLine("ToUpper - converte todo os
  caracteres para maiúsculo");
            WriteLine($"Converte para maiúsculo: {empr
esa.ToUpper()}");
        }
     }
  }
```

Salve e execute para visualizar o resultado.

***Resultado:***

**—— Funções de Textos ——**

**TRIM - retira os espaços em branco antes e após a expressão**
**Nome sem espaços: Microsoft Corporation**
**Length - retorna a qtde de caracteres**
**Tamanho do texto: 25**
**Tamanho do texto após o TRIM(): 21**
**ToUpper - converte todo os caracteres para maiúsculo**
**Converte para maiúsculo: MICROSOFT CORPORATION**

## TOLOWER

O método *ToLower* faz o contrário do *ToUpper*, ou seja, transforma toda a expressão em letras minúsculas. Se aplic também à expressões de caracteres ou variáveis. O uso se dá em casos de ler informações de qualquer fonte e formatá-la para minúsculo.

No código a seguir, adicione as linhas para uso do *ToLower*.

```
class funcoesTexto
{
    static void Main(string[] args)
    {
        WriteLine("------ Funções de Textos -------");
        ...
        WriteLine("ToUpper - converte todo os caracteres para maiúsculo");
        WriteLine($"Converte para maiúsculo: {empresa.ToUpper()}");

        WriteLine("ToLower - converte todo os caracteres para minúsculo");
        WriteLine($"Converte para minúsculo: {empresa.ToLower()}");
    }
}
```

Salve e execute o código. Para efeito de repetição de resultado, ocultei algumas linhas.

***Resultado:***

**—— Funções de Textos ——**

**ToUpper - converte todo os caracteres para maiúsculo**
**Converte para maiúsculo: MICROSOFT CORPORATION**
**ToLower - converte todo os caracteres para minúsculo**
**Converte para minúsculo: microsoft corporation**

É importante ressaltar que aos olhos da linguagem, comparar expressões com conteúdos iguais nem sempre o resultado é o esperado, por exemplo, no código a seguir qual será o resultado da comparação das variáveis nomeUpper e nomeLower?

```
static void Main(string[] args)
{
    ...

    var nomeUpper = "AIRTON SENNA";
    var nomeLower = "airton senna";
    // comparação 1
    if (nomeUpper == nomeLower)
        WriteLine("1 - nomes iguais");
    else
        WriteLine("1 - nomes diferentes");

    // comparação 2
    if (nomeUpper.ToLower() == nomeLower)
        WriteLine("2 - nomes iguais");
    else
        WriteLine("2 - nomes diferentes");

    // comparação 3
    if (nomeUpper.Equals(nomeLower, StringComparis
on.OrdinalIgnoreCase))
        WriteLine("3 - nomes iguais");
    else
        WriteLine("3 - nomes diferentes");
}
```

Salve, execute e veja os resultados. Comparar maiúsculas e minúsculas são conjuntos diferentes (resultado 1), exceto se convertermos tudo para *ToLower* e comparar, aí sim o resultado será igual (resultado 2). Já no caso do resultado 3 estamos comparando das duas variáveis (.Equals), no entanto, está explícito para ignorar maiúsculas e minúsculas. O uso do *StringComparison* permite definir o

tipo de comparação, neste caso o *OrdinalIgnoreCase*.

É importante ressaltar que na interação com usuários em mecanismos de pesquisas, convém converter tudo para ToLower ou ToUpper e depois comparar, senão o resultado será sempre diferente. Ou ainda, definir o tipo de comparação com o *StringComparison.* Imagine pesquisar num banco de dados expressões por nomes, endereços, etc.

***Resultado:***

**ToUpper - converte todo os caracteres para maiúsculo**
**Converte para maiúsculo: MICROSOFT CORPORATION**
**ToLower - converte todo os caracteres para minúsculo**
**Converte para minúsculo: microsoft corporation**
**1 - nomes diferentes**
**2 - nomes iguais**
**3 - nomes iguais**

## REMOVE

O método *Remove* serve para capturar uma quantidade de caracteres à esquerda de uma expressão, por exemplo, ler apenas os primeiros 10 caracteres. No nosso código, adicione as linhas a seguir para testarmos o *Remove*. Note que o *Remove* está sendo usado na variável empresa, o qual está declarado que apenas os 9 primeiros caracteres devem ser mostrados.

Em seguida, criamos um *array* de *nomes* com 3 nomes completos, como se fosse a lista de empregados. Num email de comunicado precisamos referenciar apenas o primeiro nome de cada um. Para isto, criamos um *looping* do tipo *foreach* para percorrer cada um dos nomes e no *WriteLine* encadeamos o *Remove* com o *IndexOf*.

Isto é bem comum, pois o *Remove* precisa saber quantos caracteres deve extrair à esquerda. E, como cada primeiro nome tem

um tamanho diferente, como dizer ao *Remove* a exata posição. Para isto, usamos o método *IndexOf* que retorna a posição exata de onde encontrar o espaço em branco, neste caso, *IndexOf(" ").*

Ou seja, a cada iteração no looping ele pega o nome completo, pesquisa a posição do espaço em branco, por exemplo 8 no primeiro nome, e usa para extrair os 8 primeiros caracteres.

```
static void Main(string[] args)
{
    ...

    WriteLine("Remove - extrai x caracteres a partir
da esquerda da expressão");
  WriteLine($"texto esquerdo: {empresa.Remove(9)}");

    WriteLine("Captura apenas o primeiro nome das
pessoas");
    string[] nomes = {"Fabricio dos Santos", "José da
Silva", "Roberta Brasil"};
    foreach(var n in nomes)
    {
        WriteLine($"{n.Remove(n.IndexOf(" "))}");
    }
}
```

Salve e execute o código. No resultado observe que o texto extraído contém apenas a palavra Microsoft, assim como apenas os primeiros nomes dos empregados.

*Resultado:*

Remove - extrai x caracteres a partir da esquerda da expressão
texto esquerdo: Microsoft
Captura apenas o primeiro nome das pessoas
Fabricio
José
Roberta

Veja na figura a seguir, na linha 50 definimos um *Breakpoint* para que a execução pare nesta linha. Abra a janela de *Debug* para mostrar as variáveis locais, neste caso, o array de nomes. Em *WATCH*, clique no ícone + para adicionar uma expressão, digite *n.IndexOf(" ")* e tecle ENTER. Quando o código para na linha 50, pressione F10 para continuar a execução passo a passo. Ao entrar no looping, na linha 54, a expressão no *WATCH* é mostrada em que posição o espaço em branco foi encontrada, o qual será usada como parâmetro do *Remove*.

Figura 14 – Debug do código

## REPLACE

O método *Replace* é usado para substituir cadeias de expressões de textos. Existem dois parâmetros, o primeiro indica qual o texto a ser substituído e o segundo é o texto que substituirá. Um uso co-

mum é em casos de tratamento de dados, por exemplo, numa lista de endereços, trocar os termos “R.”, “RUA” ou “Street” por “Rua”.

No SVM digite as linhas a seguir para trocar o nome Microsoft por Google na nova variável *novaEmpresa*. Uma vez executado o *Replace*, passa a valer este novo conteúdo para qualquer código após esta operação.

```
static void Main(string[] args)
{
   ...
   WriteLine(“Replace - troca o conteúdo da expressão”);
   WriteLine($”texto atual: {empresa}”);
   var novaEmpresa = empresa.Replace(“Microsoft”, “Google”);
   WriteLine($”texto trocado: {novaEmpresa}”);
}
```

Salve e execute o código para ver o resultado.

***Resultado:***

**Replace - troca o conteúdo da expressão**
**texto atual: Microsoft Corporation**
**texto trocado: Google Corporation**

## SPLIT

O método *Split* é muito usado em situações de tratamento de dados oriundos de arquivos textos, CSV, Excel, onde é preciso separar as cadeias de caracteres através de um caractere chave. O mais usado é o espaço em branco, mas pode ser qualquer caractere.

No exemplo de código a seguir temos uma expressão na variável *NivelLivro* e um array de *string* na variável *blocos*. O objetivo é usar o *Split* em toda a expressão de *NivelLivro*, e cada bloco de texto será

adicionado ao array “blocos”.

Em seguida, é feito um looping foreach para exibir cada conjunto de texto e ao final o método Count() conta a quantidade de itens do array.

Toda vez que o texto for exibido, a expressão contém a variável contador para mostrar a ordem de classificação. A cada iteração do *looping*, o contador é incrementado (contador++).

```
static void Main(string[] args)
{
    ...
     WriteLine(“Split - divide e extrai cada palavra em
um array”);
    string NivelLivro = “Este livro é básico de C#.”;
    string[] blocos = NivelLivro.Split(‘ ‘);
    var contador = 1;
    foreach (var exp in blocos)
    {
        WriteLine($”texto {contador++}: {exp}”);
    }
    WriteLine($”Qtde de palavras: {blocos.Count()}”);
}
```

***Resultado:***

**Split - divide e extrai cada palavra em um array**
**texto 1: Este**
**texto 2: livro**
**texto 3: é**
**texto 4: básico**

```
texto 5: de
texto 6: C#.
Qtde de palavras: 6
```

## SUBSTRING

O método Substring é usado para extrair parte do texto a partir de uma posição inicial. O tamanho do texto a ser capturado pode ou não ser informada, caso não seja, todo o texto a partir da posição é capturado. Caso declarado o tamanho, é capturado o texto conforme o número de caracteres.

Veja no código a seguir o conteúdo da variável NivelLivro que contém toda uma expressão. Na sintaxe NivelLivro.Substring(5, 14) informa que o texto a ser capturado deverá iniciar na posição 5 e pegar os próximos 14 caracteres. O resultado será “livro é básico”.

Já no *array string[ ] cesta* onde temos 4 elementos (itens da cesta de compra), como fazer para ler apenas os nomes das frutas, excluindo-se as quantidades? Criamos um *looping foreach* e a cada iteração precisamos saber qual a posição do primeiro espaço em branco *p.IndexOf(“ ”).* A soma +1 indica a próxima posição do texto, ou seja, o nome da fruta logo após o número. Assim já temos a posição inicial. E como a quantidade de caracteres não foi especificada, o método *Substring* retorna todo o restante. Em outras palavras, o código pega qualquer nome de fruta após o número.

```
static void Main(string[] args)
{
    string NivelLivro = “Este livro é básico de C#.”;
    ...
     WriteLine(“Substring é usado para extrair parte d
o texto”);
    WriteLine(NivelLivro.Substring(5, 14));

    string[] cesta = {“5 Laranjas”, “10 Goiabas
vermelhas”, “5 Pêssegos doces”, “5 Bananas”};
```

```
    foreach(var p in cesta)
    {
        // p.IndexOf(" ") +1 retorna a posição inicial
logo após o número
        // ex: 5 Laranjas = posição 1+1 = 2
        WriteLine($"{p.Substring(p.IndexOf(" ") +1 )}");
    }
}
```

Para um melhor entendimento, na linha 80 coloque um *Breakpoint* e execute F5. Na janela de *Debug*, em *WATCH* clique no ícone + e adicione as 5 expressões, conforme a figura a seguir. Quando a execução parar na linha 80, pressione F10 para executar a linha e observe em *WATCH* o resultado. Em *p.IndexOf(" ")* retorna a posição 1, onde encontrou o espaço em branco; em p.IndexOf(" *") + 1* temos 2, pois apenas somou +1; em p (variável declarada no *foreach* que representa o item em si) retorna o texto completo "5 Laranjas"; em *p.Substring(p.IndexOf(" "))* retorna o texto com o espaço em branco antes da fruta, neste caso " Laranjas"; em *p.Substring(p.IndexOf(" ") + 1)* somamos +1 na posição, então o retorno será o nome da fruta em si.

O legal de usar o *Debug* é que podemos entender exatamente o que o código está fazendo passo a passo, testar expressões em tempo de execução e corrigir, se necessário.

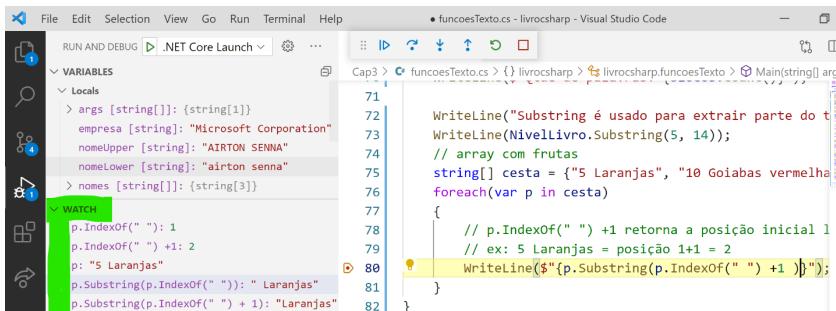



Figura 15 – Janela de Debug com os resultados

Veja o resultado final com todos os itens da cesta mostrando apenas os nomes das frutas.

***Resultado:***

**Substring é usado para extrair parte do texto**
**livro é básico**
**Laranjas**
**Goiabas vermelhas**
**Pêssegos doces**
**Bananas**

Agora aqui vai um desafio a você leitor: Como fazer para somar todas as quantidades contidas dos produtos da cesta?

Partindo do pressuposto que as quantidades estão sempre no prefixo do nome dos produtos e há um espaço em branco entre o número e o texto, criamos a variável *Qtde* do tipo *Inteiro* que durante a iteração do looping é somada a Qtde de cada produto.

```
static void Main(string[] args)
{
    ...

    string[] cesta = {"5 Laranjas", "10 Goiabas
vermelhas", "5 Pêssegos doces", "5 Bananas"};
    int qtde = 0;
    foreach(var p in cesta)
    {
        // p.IndexOf(" ") +1 retorna a posição inicial log
o após o número
        // ex: 5 Laranjas = posição 1+1 = 2
        WriteLine($"{p.Substring(p.IndexOf(" ") +1 )}");

        // Ler apenas os números para somar na variáv
el qtde
            qtde += Convert.ToInt32(p.Substring(0, p.
```

```
IndexOf(" ")));
        }
        WriteLine($"Qtde total: {qtde:n0}");
    }
```

A melhor forma de explicar este código que soma as quantidades (linha 83) é usar o *Debug* com as expressões no *WATCH*. Note que a variável P contém o texto completo do produto. O que precisamos é extrair os primeiros números apenas, converter para número inteiro e somar na variável Qtde. Então, o *p.Substring(0, p.IndexOf(" "))* retorna o número em si, pois inicia na posição zero e captura tudo o que estiver até o primeiro espaço em branco. Em seguida usamos o *Convert.ToInt32* para converter o texto que contém o número em si para inteiro, a fim de ser somado à variável Qtde.

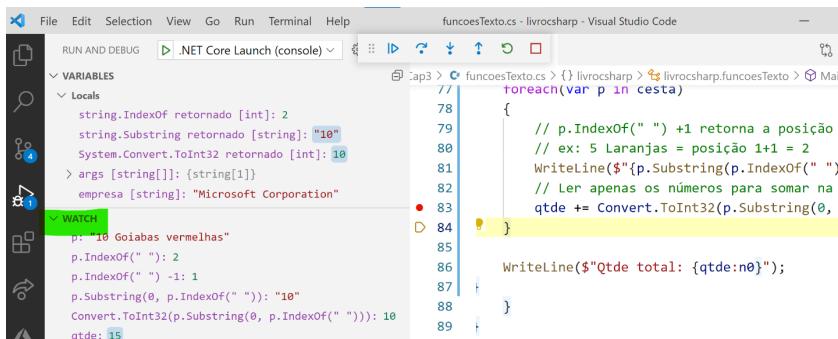



**Figura 16 – Debug para somar QTDE**

Veja que no final é mostrada a variável Qtde com total de 25.

***Resultado:***

**Laranjas**
**Goiabas vermelhas**
**Pêssegos doces**
**Bananas**
**Qtde total: 25**

## ISNULLOREMPTY

O método *IsNullOrEmpty* verifica se uma *String* está nula ou vazia. Isto é muito usado onde há interação com o usuário, afim de consistência de dados. Em casos de manipular um objeto entre as camadas de acesso a dados e interface, também usamos com frequência.

Vamos a um exemplo prático o qual temos duas variáveis, nome e sobrenome. Em seguida há um ***IF*** questionando se a variável Nome é diferente de nula ou vazia. Na sintaxe, note que o uso do **!** (exclamação) antes do *String.IsNullOrEmpty* indica a negativa.

Já o uso do && significa E (*AND*) na condição, ou seja, se as variáveis Nome e Sobrenome forem diferentes de nulo ou vazia, mostre o nome completo. Caso contrário, mostre apenas o nome.

Neste caso, poderíamos ter usado apenas um &, o que também indica o E (*AND*). Mas, no C# quando temos mais de uma condição a ser avaliada é melhor usar dois &&. Isto porque quando a primeira condição não atende ao critério, as demais não são nem validadas. Já o uso de um &, todas as condições são avaliadas, mesmo que as anteriores sejam falsas.

```
static void Main(string[] args)
{
    ...

     WriteLine("IsNullOrEmpty verifica se a string está
nula ou vazia");
    string nome = "Renato";
    string sobrenome = null;
         if (!String.IsNullOrEmpty(nome) && !String.
IsNullOrEmpty(sobrenome))
    {
         WriteLine($"Nome completo: {nome} {sobreno
me}");
```

```
    }
    else{
        WriteLine($"Nome: {nome}");
    }
}
```

Salve e execute F5 o código para visualizar o resultado.

***Resultado:***

**IsNullOrEmpty verifica se a string está nula ou vazia**
**Nome: Renato**

## FUNÇÕES DATAS

As funções de datas permitem manipular qualquer informação de uma data que esteja no modelo *DateTime* contendo o dia, o mês e o ano. Sendo assim, conseguimos fazer contas com datas, adicionar ou subtrair, dias, meses e anos, aplicar formatações customizadas, obter a data e hora completa do sistema operacional, converter em texto, ler um texto e converter em data, entre outras.

Vamos adicionar o arquivo funcoesDatas.cs. Como sempre, digite a lista de USING, o namespace, o nome da classe e o SVM (static void Main).

```
using static System.Console;
using System;
using System.Linq;

namespace livrocsharp
{
    class funcoesDatas
    {
        static void Main(string[] args)
        {
```

```
        }
    }
}
```

Agora que temos mais um arquivo .CS com o mesmo método Main no SVM *(static void Main)* e o C# precisa saber qual Main executar, é necessário configurar o arquivo **livrocsharp.csproj**. Abra-o, localize a linha do <*StartupObject*> e troque o nome para funcoesDatas, neste caso, **livrocsharp.funcoesDatas**. Ao final, salve o arquivo CTRL + S.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
          <StartupObject>livrocsharp.funcoesDatas</
StartupObject>
    </PropertyGroup>
</Project>
```

### DATETIME

Para o C# uma data, hora, minuto, segundos e ticks dependem do tipo *DateTime*. Todos estes são conhecidos, exceto o *Ticks*, que é um tipo *Long* que permite atribuir um número para milissegundos e nanosegundos.

O *DateTime* para uma data comum é composto pelo dia (1-31), mês (1-12) e o ano, todos valores numéricos. A ordem de entrada sempre será o Ano, Mês, Dia, nesta ordem respectivamente. Vamos a um código de exemplo, dentro do SVM crie três variáveis dia, mês e ano contendo os seguintes valores, conforme código a seguir.

Note como é feita a declaração da variável dtAniversario, é do tipo *DateTime* e ao atribuir *new DateTime* os parâmetros são as variáveis ano, mês e dia, nesta ordem. Podemos também definir uma data digitando números fixos, por exemplo, veja a variável *dtFesta* =

*new DateTime(2021, 08, 25).* O que não pode é desrespeitar a ordem e os valores dos três parâmetros, senão será gerado um erro.

Uma vez tendo a data válida, partimos para a formatação. Como a exibição de uma data pode ser customizada, usamos para o **dia** a letra **d** minúscula, que pode ter de 1 a 4 letras d. Para o **mês**, usamos a letra **M** maiúscula que pode ter de 1 a 3 letras M. A letra m minúscula é usada para minutos. Já o **ano**, usamos a letra **y** minúscula que pode ter 2 ou 4 letras y.

No código a seguir teremos vários formatos customizados, neste caso estão sendo exibidas dentro de uma expressão de interpolação de *Strings*. Note que o formato vem logo a seguir do nome da variável:.

```
using static System.Console;
using System;
using System.Linq;

namespace livrocsharp
{
   class funcoesDatas
   {
      static void Main(string[] args)
      {
         int dia = 12;
         int mes = 05;
         int ano = 2021;
          DateTime dtAniversario = new DateTime(ano
, mes, dia);
           DateTime dtFesta = new DateTime(2021, 08,
25);
         WriteLine("------ Funções de Datas -------");
         WriteLine($"Aniversário: {dtAniversario}");
           WriteLine($"Aniversário: {dtAniversario:dd/
MM/yyyy}");
           WriteLine($"Aniversário: {dtAniversario:ddd
```

```
d/MMM/yyyy}");
            WriteLine($"Aniversário: {dtAniversario:dddd dd/MMMM/yyyy}");
        }
    }
}
```

Salve o projeto e execute F5 para ver o resultado. Note na linha 2 do resultado, que quando não é atribuído o formato, o DateTime é mostrado com o dia/mês/ano horas:minutos:segundos. Já com o formato customizado, podemos montar o que quisermos, inclusive repetir a letra, basta ver a última linha (:dddd dd/MMMM/yyyy).

***Resultado:***

**—– Funções de Datas —–**

**Aniversário: 12/05/2021 00:00:00**
**Aniversário: 12/05/2021**
**Aniversário: quarta-feira/mai/2021**
**Aniversário: quarta-feira 12/maio/2021**

## TODAY

A propriedade *Today* do *DateTime* retorna a data completa com o dia, mês e ano do sistema operacional. Usamos com muita frequencia em aplicações onde precisamos saber a data no dia de hoje, assim o usuário não tem como informar, a data é capturada diretamente da máquina e não existem parâmetros.

## NOW

A propriedade *Now* do *DateTime* retorna a data, hora, minutos e segundos automaticamente do sistema operacional. Em aplicações de bolsa de valores, transações em bancos e uso de medicamentos são apenas alguns exemplos de onde são utilizadas.

Veja no código a seguir como são as chamadas do *Today* e do *Now*, assim como as formatações a serem aplicadas.

```
class funcoesDatas
{
   static void Main(string[] args)
   {
      ...
      DateTime hoje = DateTime.Today;
      WriteLine("TODAY - retorna a data atual");
      WriteLine($"TODAY: {hoje:dd/MM/yyyy}");

      DateTime DataHora = DateTime.Now;
      WriteLine("NOW - retorna a data e a hora atual"
);
         WriteLine($"Data e Hora: {DataHora:dd/MM/
yyyy hh:mm:ss}");
   }
}
```

Salve e execute F5 para visualizar o resultado.

***Resultado:***

**TODAY - retorna a data atual**
**TODAY: 06/04/2021**
**NOW - retorna a data e a hora atual**
**Data e Hora: 06/04/2021 06:44:47**

## DAY / MONTH / YEAR

A partir de uma data completa é possível extrair as devidas partes como dia, mês e ano, armazenar em variáveis, fazer contas, manipular, enfim, fazer o que quiser conforme o escopo da aplicação.

Para isto, basta criar uma data completa e usar as propriedades *Day, Month e Year* do objeto *DateTime*. No código a seguir estamos extraindo da variável DataHora e mostrando cada uma das partes.

```
class funcoesDatas
{
   static void Main(string[] args)
   {
      ...
      DateTime DataHora = DateTime.Now;

      WriteLine("DAY / MONTH / YEAR - capturar o dia
, mês e ano separadamente");
      WriteLine($"Dia: {DataHora.Day}");
      WriteLine($"Mês: {DataHora.Month}");
      WriteLine($"Ano: {DataHora.Year}");
   }
}
```

Execute F5 e observe o resultado.

***Resultado:***

**Data e Hora: 06/04/2021 06:52:41**
**DAY / MONTH / YEAR - capturar o dia, mês e ano separadamente**
**Dia: 6**
**Mês: 4**
**Ano: 2021**

## MANIPULAR DATA

Em uma data *DateTime* válida podemos adicionar dias *AddDays(n)*, meses *AddMonths(n)* e anos *AddYears(n)*, basta informar o respectivo número a ser adicionado.

Vamos a um exemplo clássico de pedido de compra. No código a seguir temos a data do pedido (*dtPedido*) que captura o dia atual (*Today*), a data de vencimento (*dtVencto*) que são adicionados 35

dias à *dtPedido* (*AddDays(35))*, a data do pagamento (*dtPagto*) que são adicionados 2 meses à *dtVencto (AddMonths(2)).*

Em seguida, as datas do pedido e vencimento são exibidas com o formato customizado *dd/MM/yyyy*. E como o C# dispõe de duas formatações prontas, vamos usá-las em *dtVencto* com o formato longo (*ToLongDateString*) e o formato curto (*ToShortDateString*).

Toda data tem obrigatoriamente o dia da semana (domingo, segunda, ..., sábado), e para saber qual é o dia usamos a propriedade *DayOfWeek*, o qual está sendo aplicada à *dtVencto*.

Podemos usar um formato específico do *ToString* de acordo com a cultura definida, neste caso a cultura do Brasil pt-BR, *dtVencto. ToString("dddd", new CultureInfo("pt-BR"))*. Caso use uma cultura é preciso adicionar o *using System.Globalization*.

Já o dia do ano (*DayOfYear*) aplicado à *dtVencto* mostra quantos dias foram corridos desde o início do ano.

E, para saber quantos dias se passaram entre duas datas, usamos o *Subtract*, onde referenciamos a maior data, neste caso *dtPagto*, seguido do método *Subtract*, passando como parâmetro a data a ser comparada, *dtPedido*. Isto retorna o número de dias corridos.

```
using System.Globalization;

static void Main(string[] args)
{
    ...
    DateTime dtPedido = DateTime.Today;
    // adiciona 35 dias
    DateTime dtVencto = dtPedido.AddDays(35);

    // adicionar 2 meses
    DateTime dtPagto = dtVencto.AddMonths(2);
     WriteLine($"Pedido feito em {dtPedido:dd/MMM/
```

```
yyyy} vence em {dtVencto:dd/MMM/yyyy}");
        WriteLine($"Formatação completa: {dtVencto.
ToLongDateString()}");
          WriteLine($"Formatação curta: {dtVencto.
ToShortDateString()}");

    // dia da semana
   WriteLine($"dia da semana: {dtVencto.DayOfWeek}");
    WriteLine($"dia do semana em português: {dtVenc
to.ToString("dddd", new CultureInfo("pt-BR"))}");
      WriteLine($"Número do dia da semana: {(int)
dtVencto.DayOfWeek}");

    // dia do ano
    WriteLine($"dia do ano: {dtVencto.DayOfYear}");

    // subtrai 2 datas
    var qtdeDias = dtPagto.Subtract(dtPedido);
     WriteLine($"Entre o pedido e o pagamento foram
{qtdeDias:dd} dias");
  }
```

Salve e execute F5 para ver o resultado com as manipulações das datas.

***Resultado:***

**Pedido feito em 06/abr/2021 vence em 11/mai/2021**
**Formatação completa: terça-feira, 11 de maio de 2021**
**Formatação curta: 11/05/2021**
**dia da semana: Tuesday**
**Número do dia da semana: 2**
**dia do semana em português: terça-feira**
**dia do ano: 131**
**Entre o pedido e o pagamento foram 96 dias**

## CONVERSÕES DE DATAS

Quando pensamos em ler dados externos à aplicação, seja retornando informações de uma API, banco de dados, CSV, Textos, etc, é comum uma data vir no formato texto. Então, cabe uma conversão para deixar no formato *DateTime*, permitindo manipular de acordo com a necessidade.

Veja o código a seguir onde temos uma variável *dataTexto* do tipo *String* contendo uma data no formato dia/mês/ano, só que é um texto. Já a variável *dataTextoConvertida* está definida no formato *DateTime*. E, para converter para outro formato, independente do tipo de dado, sempre há um risco de erro e tentativa. Quem faz esta tentativa é o *TryParse* onde os parâmetros são a data no formato texto e a data no formato *DateTime*. Cabe ressaltar que a palavra chave *OUT* informa qual a variável se espera o resultado convertido.

O *TryParse* serve para muitos outros formatos e podemos entender da seguinte forma: tente converter, se der certo jogue o resultado em *OUT*; caso contrário, dará erro.

No exemplo a seguir, caso a conversão seja com sucesso, o texto exibido será “Data com conversão aceita”. Em seguida, temos a variável *dataTextoErrada* com um formato de data inválido, e na tentativa de conversão será gerado um erro.

```
static void Main(string[] args)
{
    …
    WriteLine(“Conversão de Texto para Date”);
    string dataTexto = “15/07/2021”;
    DateTime dataTextoConvertida;
    // tentativa (TryParse) de conversão de dataTexto
    // caso dê certo a saída OUT será em dataTextoCo
nvertida
    if( DateTime.TryParse(dataTexto, out dataTextoCo
nvertida))
```

```
        WriteLine("Data com conversão aceita");
    else
        WriteLine("Erro na conversão da data");

    string dataTextoErrada = "15/metade do ano/2021";
    DateTime dataTextoErradaConvertida;
    if( DateTime.TryParse(dataTextoErrada, out dataTe
xtoErradaConvertida))
        WriteLine("Data com conversão aceita");
    else
        WriteLine("Erro na conversão da data");
}
```

Salve e execute F5 com o *Breakpoint* para acompanhar o fluxo desde o início, assim conseguirá ver o erro na conversão.

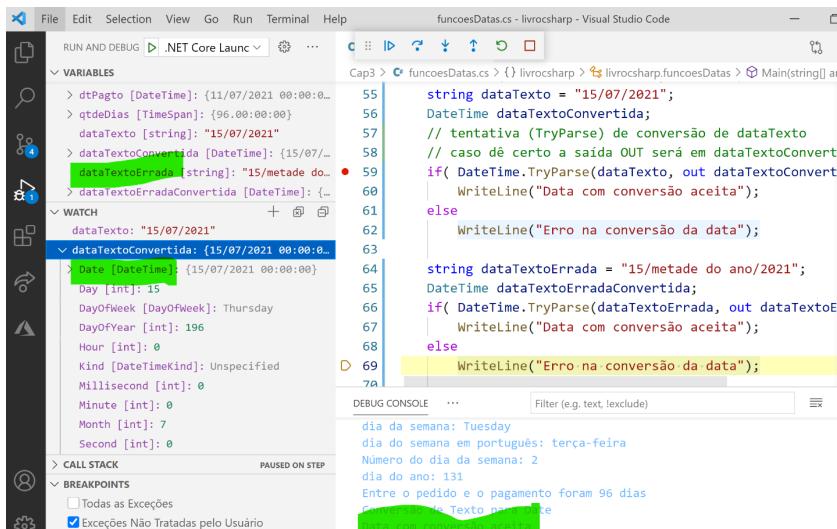



Figura 17 – Debug com das datas convertidas

Veja o resultado contendo a data convertida e a data com erro de conversão.

***Resultado:***

```
Conversão de Texto para Date
Data com conversão aceita
Erro na conversão da data
```

## DATAS EM OBJETOS

A Programação Orientada a Objetos (OOP) nos ajuda na estrutura dos objetos, entre muitos outros benefícios que não é o foco agora, temos um capítulo somente deste tópico. O importante aqui é criarmos um objeto chamado Pedido contendo as respectivas propriedades, conforme código a seguir. Todo objeto é uma classe, então na parte inferior do código, após a *class funcoesDatas*, declare uma classe pública (*public class Pedido*).

Em seguida, digite todas as propriedades do Pedido, sendo o *PedidoID* (chave do pedido), as datas *DateTime DtPedido*, *DtVencto* (que automaticamente adiciona 30 dias baseado na *DtPedido*), *DtPagto*, *DiasAtraso* (tipo *TimeSpan*) que faz a subtração entre *DtPagto* e *DtVencto*, o Valor e a Multa de 10% a ser aplicada.

Veja como que é simples definir isto no C#, basta aplicar as regras de negócio conforme a necessidade.

```
namespace livrocsharp
{
   class funcoesDatas
   {
     ...
   }

   public class Pedido
   {
      public int PedidoID { get; set; }
      public DateTime DtPedido { get; set; }
        public DateTime DtVencimento() => DtPedido.
AddDays(30);
```

```
        public DateTime DtPagto { get; set; }
            public TimeSpan DiasAtraso() => DtPagto.
Subtract(DtVencimento());
        public decimal Valor { get; set; }
        public decimal Multa => Valor * 0.10M;
    }
}
```

Agora que o objeto Pedido está definido, podemos instancia-lo no SVM. No código a seguir, atribuímos ao objeto Pedido novos dados (*new Pedido*) como *PedidoID*, *DtPedido*, *DtPagto* (45 dias a partir de *DtPedido*) e o valor. As demais *DtVencto*, *DiasAtraso* e *Multa* são calculadas automaticamente.

Em seguida, no *WriteLine* são exibidas todas as informações relativas ao pedido.

```
static void Main(string[] args)
{
    ...
    // uso de Datas em Objetos
    var pedido = new Pedido
    {
        PedidoID = 1,
        DtPedido = DateTime.Today,
        DtPagto = DateTime.Today.AddDays(45),
        Valor = 1500
    };

    WriteLine($"Pedido: {pedido.PedidoID} - " +
        $"{pedido.DtPedido:dd/MMM/yyyy} - " +
            $"vencto:  {pedido.DtVencimento():dd/MMM/
yyyy} - " +
      $"dias atraso: {pedido.DiasAtraso().TotalDays} - " +
        $"valor: {pedido.Valor:n2} - " +
        $"multa: {pedido.Multa:n2}");
}
```

Salve e execute F5 para ver o resultado. Note que todas as informações são exibidas, tanto as definidas no objeto pedido, quanto as que são calculadas automaticamente.

***Resultado:***

**Pedido: 1 - 06/abr/2021 - vencto: 06/mai/2021 - dias atraso: 15 - valor: 1.500,00 - multa: 150,00**

## CONVERSÃO DE DADOS

No C# temos dois tipos de dados que são sempre armazenados na memória, sendo tipos de **valor** e **referência**. Quando atribuímos um valor a uma variável dos tipos *int, float, double, decimal, bool e char* são do tipo **VALOR**. Isto porque o conteúdo vai diretamente para um local na memória.

Já o tipo por **REFERÊNCIA**, armazena o endereço do valor onde está armazenado, por exemplo, *object, string e array*.

Em qualquer tipo de aplicação é comum a conversão de tipos de dados, *int* para *double*, texto para data, *object* para *float* e vice-versa. A estas conversões chamamos de *Boxing* e *Unboxing*.

**Boxing** é a conversão de um tipo de valor para o tipo de objeto ou qualquer tipo de interface implementado por este tipo de valor. O ***boxing*** está **implícito**.

```
// boxing
int percentual = 10;
object objeto1 = percentual;
```

**Unboxing** é o inverso do *Boxing*. É a conversão de um tipo de referência em tipo de valor. O *unboxing* extrai o valor do tipo de referência e atribui a um tipo de valor. O *unboxing* é explícito, ou seja, precisamos declarar, por exemplo *(int) objeto2*.

```
// unboxing
object objeto2 = 10;
int desconto = (int)objeto2;
```

**Como os valores funcionam na memória do CLR (Commom Language Runtime)?**

Todas as informações são armazenadas na memória quando atribuímos valores aos objetos. O valor e o tipo de dado é apenas uma referência na memória. No exemplo acima, *int percentual* é atribuído ao *object objeto1*, sendo que *objeto1* é apenas um endereço e não um valor em si. Com isto, o CLR configura o tipo de valor criando um novo *System*.*Object* no *heap* (pilha na memória) e atribui o valor de *percentual* a ele. Em seguida, atribui um endereço desse objeto ao *objeto1*. Isto é denominado *Boxing*.

Vamos a um exemplo prático. No nosso projeto, dentro da pasta Cap3, adicione um novo arquivo chamado conversaoDados.cs, já adicione a estrutura básica, conforme o código a seguir:

```
using static System.Console;
using System;
using System.Linq;
using System.Globalization;

namespace livrocsharp
{
    class conversaoDados
    {
        static void Main(string[] args)
        {


        }
```

```
    }
}
```

Em seguida, já abra o arquivo livrocsharp.csproj e defina a nova classe no <StartupObject> para ao executar o projeto com o F5, este novo arquivo seja a fonte. Ao final, salve-o.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
        <StartupObject>livrocsharp.conversaoDados</
StartupObject>
  </PropertyGroup>
</Project>
```

Dentro do SVM (*static void Main*) digite o seguinte código para fazermos o *Boxing* de tipos numéricos *int* e *decimal*. Note que temos a variável *percentual* do tipo *int* que é atribuída à variável *objPercentual* do tipo *object*.

No código da variável *salario*, do tipo *decimal*, é atribuída à variável *objSalario* do tipo *object*. No *WriteLine* colocamos o *GetType* das variáveis para visualizarmos o tipo que o CLR definiu.

```
static void Main(string[] args)
{
    // boxing (converte um tipo para Object)
    WriteLine("----- Boxing");
    int percentual = 10;
    object objPercentual = percentual;
     WriteLine($"percentual: {percentual} - {percentua
l.GetType()}");
     WriteLine($"objPercentual: {objPercentual} - {objP
```

```
ercentual.GetType()}");

      decimal salario = 12500.50M;
      object objSalario = salario;
     WriteLine($"salario: {salario} - {salario.GetType()}");
      WriteLine($"objSalario: {objSalario} - {objSalario.
GetType()}");
   }
```

Salve e execute F5 o código. Caso queira definir um breakpoint no início e executar com o F10, é possível visualizar as conversões dos tipos conforme a execução.

***Resultado:***

**----- Boxing**
**percentual: 10 - System.Int32**
**objPercentual: 10 - System.Int32**
**salario: 12500,50 - System.Decimal**
**objSalario: 12500,50 - System.Decimal**

Veja na figura a seguir, na janela das variáveis locais, os nomes, tipos e conteúdos das mesmas.

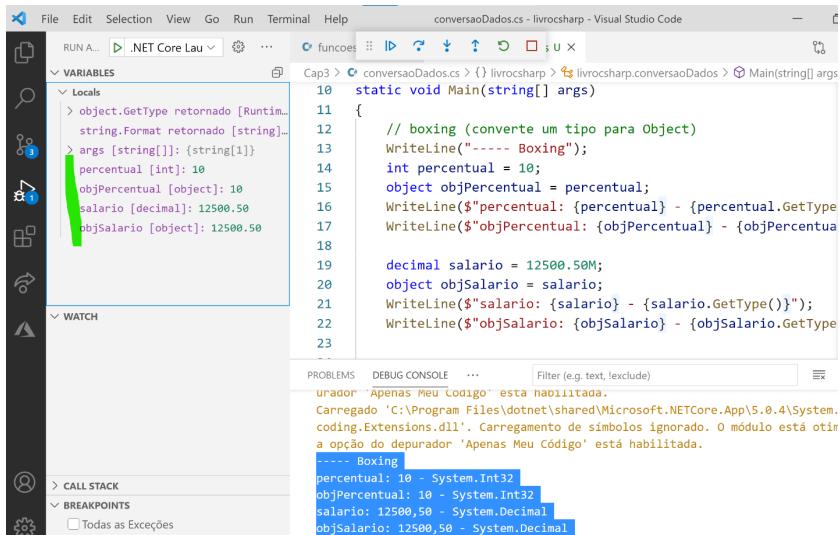



**Figura 18 – Execução do Boxing**

Agora vamos adicionar o seguinte bloco de código para fazermos o *Boxing* de tipos texto (*string*) e data (*DateTime*). A ideia é a mesma que a anterior, criamos as variáveis que serão atribuídas a um *object* e, ao final, mostramos os dados.

```
static void Main(string[] args)
{
    ...
    string nome = “Visual C#”;
    object objNome = nome;
    WriteLine($”nome: {nome} - {nome.GetType()}”);
    WriteLine($”objNome: {objNome} - {objNome.GetType()}”);

    DateTime hoje = DateTime.Today;
    object objHoje = hoje;
    WriteLine($”hoje: {hoje} - {hoje.GetType()}”);
    WriteLine($”objHoje: {objHoje} - {objHoje.GetType()}”);
```

```
}
```

Execute novamente e teremos o resultado a seguir.

***Resultado:***

**nome: Visual C# - System.String**
**objNome: Visual C# - System.String**
**hoje: 07/04/2021 00:00:00 - System.DateTime**
**objHoje: 07/04/2021 00:00:00 - System.DateTime**

Agora vamos criar um código para mostrar o *unboxing*, que é o oposto do *boxing*. Neste caso, vamos definir a variável *objDesconto* do tipo *object* que é atribuída à variável *desconto* do tipo *int*. Note na sintaxe que é obrigatório colocar entre parênteses do tipo da variável que receberá o *object*. Neste caso, usamos *(int)objDesconto*. Ao final, mostraremos o conteúdo e o tipo de dado.

```
static void Main(string[] args)
{
      ...
    // unboxing (converte um objeto para um tipo)
    object objDesconto = 10;
    int desconto = (int)objDesconto;
    WriteLine("----- Unboxing");
      WriteLine($"desconto: {desconto} - {desconto.
GetType()}");
     WriteLine($"objDesconto: {objDesconto} - {objDes
conto.GetType()}");
}
```

Execute e veja o resultado a seguir. Claro que não tivemos nenhum tipo de erro, usamos exemplos simples e reais do dia a dia de um desenvolvedor. Tenha em mente que no *unboxing* o *object* a ser atribuído precisa permitir tal conteúdo, não adianta jogar um dado *texto* em um *int* ou *double* que dará erro.

***Resultado:***

**----- Unboxing**
**desconto: 10 - System.Int32**
**objDesconto: 10 - System.Int32**

## PARSE

Uma outra forma que existe de conversão de tipos de dados é através do *Parse*. A sintaxe deve conter o tipo de campo (*Int32, Int64, int, DateTime*) seguido do *.Parse* contendo o valor a ser convertido.

Adicione o código a seguir onde temos diversos exemplos de *Int16*, *Int64*, *int* e *DateTime*. Note que quando há um texto com símbolos de negativos (parênteses ou - ), separador de milhar, símbolo da moeda R$ ou espaço em branco à esquerda ou direita do número, é preciso informar ao *Parse* do que se trata. Isto ocorre através do *Enum NumberStyles (AllowParentheses, AllowThousands, AllowCurrencySymbol, AllowLeadingSign, AllowLeadingWhite, AllowTrailingWhite).*

No exemplo a seguir, no uso do símbolo da moeda, colocamos o comando para exibir o nome da cultura atual *CultureInfo.CurrentCulture.Name* e o símbolo de moeda usado *NumberFormatInfo.CurrentInfo.CurrencySymbol*. Todo o código está dentro do bloco *try* .. *catch*, pois se ocorrer algum erro na conversão, o código cairá no catch e será mostrada uma mensagem de erro.

```
static void Main(string[] args)
{
        ...
    try
    {
        // Conversões com PARSE
        WriteLine("----- .Parse");
        WriteLine($"{Int16.Parse("150")}");
```

```
        // retorna -150, os parênteses representam o
   valor negativo
        WriteLine($"{Int16.Parse("(150)", NumberStyles.
AllowParentheses)}");

        // retorna 50000 mesmo com o ponto de milhar
         WriteLine($"{int.Parse("50.000", NumberStyles.
AllowThousands)}");

        // retorna 50000 mesmo com o símbolo
   monetário
                WriteLine($"Cultura  atual:  {CultureInfo.
CurrentCulture.Name}");
        WriteLine($"Símbolo da moeda: {NumberFormat
Info.CurrentInfo.CurrencySymbol}");
        WriteLine($"{int.Parse("R$50000", NumberStyles.
AllowCurrencySymbol)}");

        // retorna -200 mesmo com o símbolo de
   negativo -
            WriteLine($"{int.Parse("-200",  NumberStyles.
AllowLeadingSign)}");

        // retorna 200 mesmo com os espaços em
   branco antes e depois
        WriteLine($"{int.Parse(" 100 ", NumberStyles.All
owLeadingWhite | NumberStyles.AllowTrailingWhite)}");

        // retorna 5432123456
        WriteLine($"{Int64.Parse("5432123456")}");

     WriteLine($"Data:{DateTime.Parse("21/4/2021"):dd/
MMM/yyyy}");
     }
     catch (Exception ex)
     {
        WriteLine(ex);
```

```
    }
}
```

Salve e execute para visualizar o resultado. Note que todas as conversões foram feitas com sucesso. Caso você esteja executando este código em um computador que não esteja na cultura pt-BR, troque o símbolo da moeda para a cultura do seu computador. Todos os números estão sem formatação, exceto a data que foi especificada uma formatação.

***Resultado:***

**—— .Parse ——**

**150**
**-150**
**50000**
**Cultura atual: pt-BR**
**Símbolo da moeda: R$**
**50000**
**-200**
**100**
**5432123456**
**Data: 21/abr/2021**

## CONVERT.TO

Há mais uma forma de conversão de tipos que usamos no C#, chama-se *Convert.To(tipo).* A sintaxe é simples, basta escrever o *Convert.To+tipo* de dado, que pode ser *Int16*, *Int32*, *Int64*, *Decimal, String, Boolean ou DateTime*.

Adicione os códigos a seguir para entendermos melhor. Na parte de conversão de números, especificamos os tipos só para deixar mais claro, porém poderíamos ter usado o ***var*** (modo implícito). No caso de números tudo depende do tamanho do mesmo, podendo ser inteiro, longo ou decimal. Nos exemplos, todas as linhas mos-

tradas no resultado usam o *GetType()* só para provar o tipo, na prática não se usa isto.

Já na conversão para *String* é passado um valor decimal (250.59M), sem problema algum. Aliás a conversão para *String* é a mais fácil pois praticamente tudo pode ser convertido para *String*. No exemplo da variável texto1, no final temos um questionamento ?? se o valor é nulo ou não. Esta sintaxe do ?? é fantástica, pois o compilador já pergunta se o valor de texto1 é nulo. Caso negativo é exibido o próprio valor de texto1; caso afirmativo, é mostrado o "texto é nulo". Esta sintaxe do C# economiza um código de *IF().*

No exemplo do tipo boleano (verdadeiro ou falso), a conversão permite ler valores *false*, *true* ou números, sendo que o zero (0) é falso.

Já na conversão para *DateTime*, a variável natal recebe um texto com o dia/mês/ano e é convertido para o tipo *DateTime*. Em seguida, usamos as propriedades *Day*, *Month* e *Year* para mostrar os respectivos dados. Se não estivesse no formato de data, ocorreria um erro.

```
static void Main(string[] args)
{
      ...
   try
   {
   // Conversões com Convert.To
   WriteLine("----- Convert Números");
   int n1 = Convert.ToInt16("100");
   WriteLine($"{n1.GetType()} - {n1}");
   Int32 n2 = Convert.ToInt32("200500");
   WriteLine($"{n2.GetType()} - {n2}");
   Int64 n3 = Convert.ToInt64("100323213132132 1");
   WriteLine($"{n3.GetType()} - {n3}");
   decimal preco = Convert.ToDecimal("1420,50");
   WriteLine($"{preco.GetType()} - {preco:n2}");
```

```
    WriteLine("----- Convert String");
    string texto1 = Convert.ToString(250.59M);
                                        WriteLine($"{texto1.
GetType()} - {texto1} - resultado: {texto1 ?? "texto
é nulo"}");

    string texto2 = Convert.ToString(DateTime.Today);
    WriteLine($"{texto2.GetType()} - {texto2}");

    WriteLine("----- Convert Boleano");
    bool valido = Convert.ToBoolean("false");
    WriteLine($"{valido.GetType()} - {valido}");
    WriteLine($"0 - {Convert.ToBoolean(0)}");
    WriteLine($"1 - {Convert.ToBoolean(1)}");
    WriteLine($"100 - {Convert.ToBoolean(100)}");

    WriteLine("----- Convert Data");
   DateTime natal = Convert.ToDateTime("25/12/2021");
      WriteLine($"Natal: {natal.GetType()} - {natal:dd/
MMMM/yyyy}");
            WriteLine($"Natal: {natal.Day} - {natal.
Month} - {natal.Year}");

    }
    catch (Exception ex)
    {
       WriteLine(ex);
    }
  }
```

Salve e execute F5. Neste exercício também aconselho usar o *Debug* para investigar os valores durante as conversões.

***Resultado:***

**—— Convert Números ——**

```
System.Int32 - 100
System.Int32 - 200500
System.Int64 - 1003232131321321
System.Decimal - 1.420,50

— Convert String —

System.String - 250,59 - resultado: 250,59
System.String - 07/04/2021 00:00:00

— Convert Boleano —

System.Boolean - False
0 - False
1 - True
100 - True

— Convert Data —

Natal: System.DateTime - 25/dezembro/2021
Natal: 25 - 12 - 2021
```

# CAPÍTULO 4 – COLEÇÕES MAIS COMUNS UTILIZADAS NO C#

Neste capítulo, veremos as coleções mais comuns utilizadas no C#. As coleções nas linguagens de programação são basicamente estruturas de dados que podemos criar ou gerenciar quando queremos definir objetos e utilizar os mesmos em nossos sistemas. A ideia central de utilizar coleções em uma linguagem de programação é representar aspectos do mundo real no seu sistema. Por exemplo, imaginemos que precisamos criar no sistema um controle de senhas para gerenciar uma recepção de um consultório.

Neste consultório teremos apenas cem vagas disponíveis por dia apenas. Como representaremos isso no sistema então? Precisaremos de uma coleção de um tipo de dados para gerenciar esse controle de senhas. Como dito anteriormente, existem diversos tipos de coleções e cada uma delas é adequada para um determinado cenário.

Sendo assim, veremos as principais coleções e como utilizá-las no dia a dia de desenvolvimento de softwares.

## ARRAYS

As arrays são normalmente utilizadas e criadas quando temos um número fixo de posições em mente. Como no exemplo citado, temos apenas cem senhas disponíveis, então podemos criar e declarar o array da seguinte forma:

```
Declarando uma array com cinco elementos do tipo inteiro
int[] primeiraArray = new int[100];
```

A declaração acima é bem simples, poderíamos guardar apenas cem números do tipo inteiro nessa array. Perceba que estamos apenas deixando disponível, mas não atribuindo os valores ao array. Para o caso de já iniciar uma array com os valores atribuídos, usamos:

```
// Declarando uma array e já atribuindo valores
int[] segundaArray = new int[] { 1, 3, 5, 7, 9 };
```

Veja acima que ao invés de informar para o C# que temos cem posições de inteiros disponíveis, estamos apenas informando que temos posições para alocar os números e já colocando os valores diretamente na criação.

Paralelamente, se você não quiser informar o tipo (neste caso, int), também é possível iniciar dessa forma:

```
// Forma alternativa de criar array
int[] terceiraArray = { 1, 2, 3, 4, 5, 6 };
```

Essas formas de criação de array acima vão basicamente criar o que chamamos de arrays dimensional. Teremos apenas uma dimensão para alocar e trabalhar com os valores.

Existe também uma forma de criar uma array multidimensional, como se fosse uma matriz. Nesse caso, na criação da array, precisamos informar na declaração, conforme exemplo:

```
// Declarando uma array multi-demensional
int[,] arrayMultiDimensional = new int[2, 3];
```

Esse seria um cenário um pouco mais avançado, mas ainda válido. Para atribuir valores já na criação de uma array multidimensio-

nal, poderíamos fazer da seguinte forma:

```
// Declarando uma array multi-demensional e atribuindo valores
[,] arrayMultiDimensional2 = { { 1, 2, 3 }, { 4, 5, 6 } };
```

Esse seria um cenário um pouco mais avançado, mas ainda válido. Para atribuir valores já na criação de uma array multidimensional, poderíamos fazer da seguinte forma:

## ACESSANDO O VALOR DE UMA ARRAY

Até aqui já criamos as arrays e atribuímos valores na sua criação. Porém, em um cenário real de desenvolvimento de softwares, precisamos obter os valores que estão contidos nessas arrays para exibir em alguma tela, por exemplo:

```
using System;

namespace livrocsharp {

   public class ExemploClasseArray
   {
   static void ClasseArray()
      {
         // Declarando uma array e já atribuindo valores
         int[] segundArray = new int[] { 1, 3, 5, 7, 9 };

         Console.Write("Valor da array na posição 0 -> {0}  ", segundArray[0]);
      }
   }
}
```

Esse código acima pode ser encontrado no projeto do Github, dentro da pasta Cap4. A execução desse código vai retornar o se-

guinte resultado:

**"Valor da array na posição 0 -> 1"**

O motivo disso é porque quando vamos ler uma array, iniciamos pelo indexador zero.

```
Arrays.cs U ×
Cap4 > Arrays.cs > {} ExemploArrays
1   using System;
2
3   namespace ExemploArrays {
4
        0 references
5       public class ExemploClasseArray
6       {
            0 references
7           static void Main()
8           {
9               // Declarando uma array e já atribuindo valores
10              int[] segundArray = new int[] { 1  3, 5, 7, 9 };
11
12              Console.Write("Valor da array na posição 0 -> {0}  ", segundArray[0]);
13          }
14      }
15  }
```

Figura 19 - Posição de uma array

Veja na figura anterior que o valor 1 está na posição 0. O valor 3 está na posição 1. O valor 5 está na posição 3 e assim por diante. Em resumo, a posição de uma array inicia em zero e vai até o tamanho definido da array. Veja a lista completa como ficaria:

- Posição 0 - valor 1
- Posição 1 - valor 3
- Posição 2 - valor 5
- Posição 3 - valor 7
- Posição 4 - valor 9

Pode até parecer confuso, mas não é. O motivo de iniciar do zero remete a forma como os computadores foram projetados e essa convenção é adotada até hoje em estrutura de dados.

## LISTAS

As listas no C# têm a finalidade de armazenar um tipo de dados no sistema. Esses tipos de dados comumente são denominados como listas tipadas. Ao se referir a “tipadas”, estamos meramente informando ou entendendo que ao criar uma lista com um tipo de dados, como por exemplo *string*, todos os valores daquela lista devem ser de fato uma *string*. A tipagem garante que estamos trabalhando de forma efetiva e não estamos misturando tipos de dados diferentes na mesma coleção.

As listas são utilizadas normalmente em cenários onde mais de um valor deve ser associado a uma variável. Pense num cenário onde precisamos criar uma estrutura de dados para salvar o nome de todos os funcionários de um determinado departamento de uma empresa. Nesse departamento temos a Maria, o João, o André e a Flávia. Dessa forma, precisaremos criar uma estrutura de dados do tipo *List*<*T*> para acrescentar os nomes e manipular essa estrutura de dados, conforme exemplo a seguir:

```
using System;
using System.Collections.Generic;

namespace livrocsharp {

   public class ExemploList
   {
   static void Testa()
      {
          List<string> nomesFuncionarios = new List<s
tring>();

         nomesFuncionarios.Add(“Maria”);
         nomesFuncionarios.Add(“João”);
         nomesFuncionarios.Add(“André”);
         nomesFuncionarios.Add(“Flávia”);
```

```
            Console.WriteLine();
            foreach(string pessoa in nomesFuncionarios)
            {
                Console.WriteLine(pessoa);
            }

//Console.WriteLine(nomesFuncionarios[0]);
        }
    }
}
```

**O resultado desse processamento será o seguinte:**

**Output:**
**Maria**
**João**
**André**
**Flávia**

A seguir, vamos analisar em detalhes o que fizemos. Criamos uma instância de uma lista de funcionários utilizando o *List*<*string*>.

```
List<string> nomesFuncionarios = new List<string>();
```

Dessa forma, estamos criando na memória uma estrutura para guardar uma lista do tipo *string*. Se tentarmos incluir nessa lista um valor inteiro, por exemplo 0 ou 1, não será possível, pois ela foi tipada com o tipo *string*.

Por se tratar de uma lista, podemos manipular os valores da mesma, acrescentando ou removendo valores. Para isso temos o método Add() que vai possibilitar incluir uma nova *string* nessa lista, conforme o exemplo a seguir:

```
nomesFuncionarios.Add("Maria");
```

```
nomesFuncionarios.Add("João");
nomesFuncionarios.Add("André");
nomesFuncionarios.Add("Flávia");
```

A forma como a lista é indexada, ou seja, seus valores são determinados e iniciando em 0, então, se tentarmos obter o primeiro valor da lista, precisamos programar da seguinte forma:

```
Console.WriteLine(nomesFuncionarios[0]);
```

A saída desse processamento será "Maria" pois Maria é o primeiro valor que foi adicionado na lista.

Podemos ter cenários onde será necessário remover pessoas da lista. Nesse caso temos o método RemoveAt() se soubermos qual a posição exata da pessoa na lista. No exemplo a seguir vamos remover a Maria novamente da lista e já sabemos que ela é a primeira da lista, sendo assim a posição dela é 0. Podemos fazer da seguinte maneira:

```
using System;
using System.Collections.Generic;

namespace livrocsharp {

   public class ExemploList
   {
   static void Testa()
      {
         List<string> nomesFuncionarios = new List<string>();

         nomesFuncionarios.Add("Maria");
         nomesFuncionarios.Add("João");
         nomesFuncionarios.Add("André");
         nomesFuncionarios.Add("Flávia");
```

```
            Console.WriteLine();
             // foreach(string pessoa in nomesFuncionari
os)
            // {
            //     Console.WriteLine(pessoa);
            // }

            //Console.WriteLine(nomesFuncionarios[0]);
            //removendo Maria da lista
            nomesFuncionarios.RemoveAt(0);
            Console.WriteLine(nomesFuncionarios[0]);
        }
    }
}
```

**O resultado final do processamento acima será:**

**João**

Esse resultado foi executado pois acrescentamos os quatros nome na lista, mas em seguida fizemos a remoção do indexador zero, o qual representava o valor *Maria*. Sendo assim, havia quatro nomes na lista e removemos o primeiro. Nesse mesmo momento, *João* passou a ser o indexador zero que anteriormente estava ocupado por *Maria*.

Esse tipo de manipulação de dados é muito comum e corriqueiro no desenvolvimento de softwares. As manipulações de dados e estrutura de dados representam boa parte do cotidiano de um de uma pessoa desenvolvedora de softwares.

## TIPOS GENÉRICOS

Nos exemplos anteriores fixamos que a lista era do tipo *string*. Esse é um bom cenário quando temos certeza que os tipos de valores são exatamente os que precisaremos para trabalhar. Porém,

em algumas situações precisaremos flexibilizar essa tipagem dos valores.

Imaginemos um cenário onde precisamos colocar em uma mesma lista valores inteiros e letras, ou melhor, criar uma lista apenas de inteiro e outra apenas de letras. Nesse caso, o C# tem disponível o conceito de genéricos.

Os tipos genéricos facilitam esse cenário e tudo que precisamos fazer é utilizar parâmetros do tipo genérico **T**. Podemos então escrever uma única classe que qualquer outra parte do sistema pode utilizar sem precisar criar uma nova estrutura e passando o tipo específico que ela deseja.

Para ficar mais claro, vamos utilizar o seguinte exemplo. Imaginamos que queremos criar uma classe que vai possibilitar a criação de várias listas de qualquer tipo, ou seja, *strings*, *inteiro* ou até mesmo de outra classe. Nesse cenário, seria mais produtivo criarmos uma classe genérica que nos possibilite passar como parâmetro um tipo qualquer. Veja o exemplo de código:

```
using System;

namespace livrocsharp {

   // Declarando uma classe genérica
   public class ListaGenerica<T>
   {
      public void Adicionar(T input) { }
   }
   class TestListaGenerica
   {
      private class ExampleClass { }
      static void Testa()
      {
         // Declarando uma lista do tipo inteiro
          ListaGenerica<int> lista1 = new ListaGeneric
```

```
a<int>();
            lista1.Adicionar(1);

            // Declarando uma lista do tipo string
             ListaGenerica<string> lista2 = new ListaGene
rica<string>();
            lista2.Adicionar("");

            /// Declarando uma lista do tipo de uma
   classe
             ListaGenerica<ExampleClass> lista3 = new Li
staGenerica<ExampleClass>();
            lista3.Adicionar(new ExampleClass());
         }
      }
   }
```

Os tipos genéricos facilitam criar as estruturas de dados de uma forma dinâmica, facilitando a centralização e reaproveitamento de código. No exemplo anterior, a classe *ListaGenerica<T>* assume essa responsabilidade de flexibilizar a tipagem de dados utilizando o parâmetro *<T>.* Dessa forma, podemos passar qualquer tipagem de dados e utilizar essa classe em qualquer local do sistema de forma centralizada.

# CAPÍTULO 5 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM C#

Quando aprendemos programação, na maioria das vezes é de maneira estruturada, ou seja, efetuar a leitura de um código de "cima para baixo", começando no topo, executando cada linha na sequência de maneira estruturada e lógica. Claro que não há nada de errado nisso, de fato é assim que se aprende lógica de programação, tão fundamental para quem está iniciando no mundo da programação. Pensar estruturado, com lógica é fundamental, é o começo de tudo e nos ajuda em muitas decisões na vida, estar diante de um código é um problema a ser solucionado ou codificado.

Mas entendemos que este modelo de programação, ainda presente em muitas linguagens não é aplicado em C#. De fato, nunca foi, C# nasceu em 1999 (data em que foi lançado oficialmente no Brasil) orientado a Objetos. Entender o básico da orientação a objetos lhe ajudará a tirar o máximo de proveito da linguagem C#, reutilizar código, criar suas próprias bibliotecas de comando, proteger acessos indevidos a funções e comandos que são apenas internos, evitando assim, erros e bugs desnecessários. Além de tudo, como já dizia Stroustrup (O criador da linguagem de programação C++, que é o C orientado a objetos) *"OOP deixa seu código organizado, com acessos bem definidos e além de tudo elegante"*. Por que não ter um código elegante e organizado? Conseguimos isso com a orientação a objetos, e isto vai além do que elegância e organização, trata-se de qualidade de confiabilidade de código.

## LÓGICA DA ORIENTAÇÃO A OBJETOS

A Orientação a objetos possui alguns pilares importantes e essenciais de serem entendidos antes de começar a codificar. Uma rápida pesquisa na internet encontrará centenas de artigos, alguns com os 5 pilares do OOP, outros com os 12 Pilares o OOP, mas neste livro não vamos numerar os pilares, e sim discorrer agora sobre alguns conceitos importantes em programação orientada a objetos que devemos saber antes de codificar.

## PRÓXIMO AO MUNDO REAL

A Orientação a Objetos (e por isso leva o nome de “Objetos”) tenta aproximar a programação ao que vemos no mundo real. Tratando partes do código como se fossem objetos (fisicamente reais) e com suas características e propriedades, com isso conseguimos abordar um problema a ser solucionado em pequenas partes de código, que juntas solucionam o problema inteiro. Veja na figura a seguir o entendimento desta abordagem, comparando com a programação estruturada.

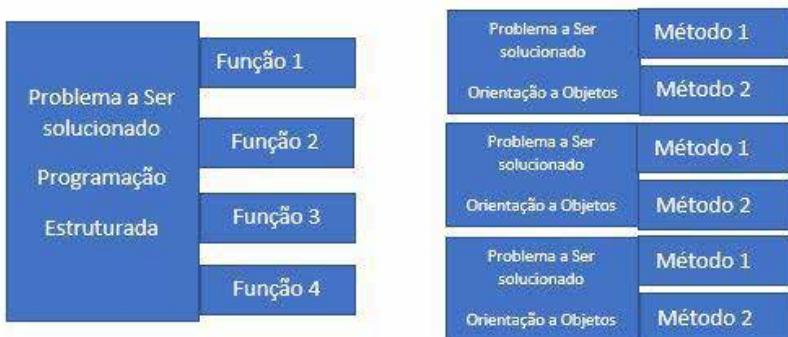



Figura 20 – OOP

Observe que a abordagem do problema a ser solucionado é diferente na orientação a objetos. Procuramos quebrar o problema em diversos “objetos” e cada objeto possui uma caraterística, um nome, uma funcionalidade e uma propriedade. Neste conceito va-

mos entender um dos principais pilares da Orientação a objetos, a Abstração.

## ABSTRAÇÃO

O conceito da abstração é tornar aquele código autossuficiente de maneira que possa ser chamado por outras partes do seu programa, sem que você necessite entender como ele foi implementado para que assim possa chamá-lo. Usamos isso em nossa vida, por isso a programação a objetos é uma maneira de aproximar a programação ao mundo real. Geralmente consumimos produtos e serviços que não sabemos exatamente como eles foram feitos, apenas utilizamos. A Abstração na orientação a objetos tem este objetivo, tornar parte de um código complexo abstraído de toda uma regra de negócio, por exemplo, e assim ser chamado quando houver necessidade diversas vezes ao longo do seu programa.

## ENCAPSULAMENTO

Este é o que consideramos ser outro pilar da programação orientada a objetos. Encapsulamento é a forma de "esconder" ao mundo uma parte do código que não faz necessário todo o programa conhecer. No gráfico da figura anterior, observe que temos Método 1 e Método 2 em cada um dos objetos criados para solucionar o problema. Mas os *problemas* não precisam enxergar a implementação um do outro, apenas saber que existem comandos (ou métodos) chamado "Método 1" e "Método 2" que irão resolver o problema proposto, sem que haja necessidade de dar acesso a todo o código implementado. Com isso, protegemos parte do código que está dentro dos métodos, contra acesso indevido. Uma vez que um código (ou objeto) é encapsulado, seus detalhes não são mais visíveis ou acessíveis ao restante do programa, apenas aquela parte do programa autorizada à visibilidade. Aqui entram os modificadores chamados "Public", "Private", "Protected" que mais adiante trataremos no código C#.

## HERANÇA

Outro pilar importante na orientação a objetos. Lembre-se do que comentamos, que a OOP tenta aproximar a codificação ao mundo real. Herança no mundo real é a passagem de bens (ativos ou não) para outra pessoa, geralmente dentro de uma linhagem hierárquica da mesma família. Uma vez que a pessoa recebeu a herança, por exemplo, uma casa ou um carro, ela pode escolher modificá-lo, mudando a cor, aumentando os cômodos. No entanto, o ponto de partida é um Ativo que ela já recebeu e assim realiza duas modificações necessárias. Na Orientação a objetos é a mesma coisa.

A herança é hierarquicamente baseada, pode ser modificada após herdada. Uma classe pode ser herdada, ou seja, ela recebe tudo o que é possível de aquele código anterior, possibilitando que ela modifique ou adicione o que for necessário para dar continuidade aquela classe (objeto código) que foi herdado. Pode parecer confuso no início, mas vamos ver isso em C#, e ao ver o código em C# vale apena voltar e ler este parágrafo.

Veja um exemplo de herança, vamos codificar um sistema que irá catalogar animais. Temos um "objeto" ou classe chamado de "Animais", o qual possui características globais de todos os animais, por exemplo, idade, peso, altura, raça, cor, etc. Se desejamos refinar um pouco mais o código, a fim de controlar os animais mamíferos, por exemplo, herdar a classe "Animais" e nela adicionar a característica "Glândulas de Mama", que só estão presentes nos animais mamíferos. Sem a necessidade de repetir novamente todas as características como idade, peso, altura, etc.

## POLIMORFISMO

Outro poderoso pilar da orientação a objetos. O Polimorfismo permite que os objetos (ou classes) possuam o mesmo nome de método, a mesma implementação, para facilitar a leitura do código e até mesmo porque às vezes a regra de negócio aplicada traz esta

necessidade de ter o mesmo nome de “Método” em classes herdadas. Possuem a mesma assinatura de comando ou de método, mas a implementação ou resultado é diferente. Já falamos de herança, então vamos aproveitar para explicar polimorfismo com herança.

Vamos ao exemplo clássico ensinado nas universidades sobre Polimorfismo. O Homem e os animais mamíferos. Imagine o desenvolvimento de um sistema relacionado a este tema, a melhor prática seria a seguinte:

Classe (ou Objeto) Mamífero – Nesta classe mamífero há uma função ou método chamado Locomocao( ). Obviamente não sabemos e não conhecemos esta locomoção porque ela é específica de cada animal. Com isso começamos a usar herança, o qual implementa o método “Locomocao ( )”, conforme a necessidade.

Sendo assim Classe (ou Objeto) Macaco herdado da classe Mamífero que possui sua função “Locomocao ( )” implementado com o resultado de “de galho em galho”.

Classe (ou Objeto) Homem herdado da classe Mamífero que possui sua função “Locomocao ( )” implementado com o resultado de “Caminhando”.

Classe (ou Objeto) Canguru herdado da classe Mamífero que possui sua função “Locomocao ( )” implementado com o resultado de “Saltando”.

```
Exemplo do código:
classe Mamifero
Locomocao() // ainda a ser codificado na herança

Mamifero ObjetoMamifero;

ObjetoMamifero = new Macaco();
```

```
ObjetoMamifero.Locomocao(); // de galho em galho

ObjetoMamifero = new Homem();
ObjetoMamifero.Locomocao(); // Caminhando

ObjetoMamifero = new Canguru();
ObjetoMamifero.Locomocao(); // Saltando
```

Sem dúvida nos exercícios que vamos propor para entender orientação a objetos em C# será mais claro o conceito de polimorfismo, não se preocupe em entender 100% o conceito nestes parágrafos.

## ENTENDENDO AS CLASSES (OBJETOS)

Vamos entender sobre classes criando a primeira classe no C# para mostrar um texto digitado pelo usuário. Vamos fazer primeiro de maneira estruturada e depois vamos abstrair os comandos dentro de uma classe, chamando-a desde o programa principal. Certifique-se que o VS Code está instalado, configurado, compilando e executando código em C#. Se isso ainda não foi feito, leia o capítulo sobre a instalação.

Abra o VS Code, crie duas pastas CAP5\OlaMundo e o arquivo Program.cs digite:

```
using System;
namespace MinhaPrimeiraClasse
{
    class ProgramaPrincipal
    {
        static void Main(string[] args)
        {
```

```
            //Declaramos uma variavel do tipo string (texto)
            String Digitado = new String("");
            //Solicitamos que o usuario digite alguma 
coisa e
            //armazene na variavel criada
            Digitado = Console.ReadLine();
            //Mostramos o que foi digitado
            Console.WriteLine(Digitado);
    }
}
```

Agora vamos transformar este código em uma classe, de maneira que ela possa ser chamada em outras partes do código sem que sejam necessárias todas as vezes chamar as funções do System.WriteLine e System.ReadLine. Sendo assim, comente o bloco interno do static void Main e substitua pelo código a seguir. Mas, primeiro crie a classe InteracaoUsuario para não apresentar erro. Neste código, abstraímos os comandos WriteLine e ReadLine, os quais são invocados na classe ProgramaPrincipal. Código muito simples, mas que já permite entender que a qualquer momento do código, podemos esta classe InteracaoUsuario.

```
using System;
namespace MinhaPrimeiraClasse
{
    class ProgramaPrincipal
    {
        static void Main(string[] args)
        {
            //Criamos o objeto Pedido herdado da clase 
InteracaoUsuario
            InteracaoUsuario Pedido = new InteracaoUsu
ario();
            //Invocamos os metodos do objeto Pedido 
herdado da classe mãe InteracaoUsuario
```

```
            var Digitado = Pedido.SolicitarDigitacao();
            Pedido.MostrarDigitacao(Digitado);
        }
    }

    //Criamos a Classe InteracaoUsuario com dois
metodos principais
    //Método que solicita a digitação de algo no
terminal
    //Método que mostra o que foi digitado
    class InteracaoUsuario
    {
        public string SolicitarDigitacao()
        {
            return Console.ReadLine();
        }
        public void MostrarDigitacao(string Digitado)
        {
            Console.WriteLine(Digitado);
        }
    }
}
```

Aqui aproveitamos um conceito interessante em programação: *sempre há como fazer diferente e melhor*. Tenho certeza que um programador experiente já olhou para este código e percebeu melhorias, as quais podem ser feitas e implementadas a qualquer momento. Vamos aproveitar e ver algumas variações de implementação da mesma classe, porém agora usando o GET e SET, protegendo assim a implementação da classe e ampliando o encapsulamento.

Os métodos GET e SET permitem acessar as propriedades privadas da classe sem que haja a necessidade de acessar diretamente aquela variável, não necessitando torná-la pública para todo o seu programa. Vamos chamar isso de controle de acesso. Este controle de acesso permite que apenas o objeto da classe derivada obtenha valores e inclua valores nas variáveis da classe através dos métodos

GET e SET.

Vamos ao exemplo com GET e SET. Na pasta CAP5, crie uma sub-pasta OlaMundo_GetSet e o arquivo Program.cs. O código a seguir abaixo é idêntico ao anterior, só que agora realizamos algumas modificações. Primeiro, abstraímos ainda mais a classe *InteracaoUsuario* fazendo com que ela não apenas receba e imprima o valor digitado no console, realizando assim a operação completa.

Segundo, criamos uma propriedade para a classe chamada *Mensagem* e através dela é que iremos recuperar a mensagem que foi digitada para futura manipulação no programa. Usamos o SET para atribuir um valor e o GET para ler o valor em si.

Terceiro, implementamos no programa principal a manipulação da mensagem que foi apresentada pela classe. Realizamos isso recebendo a mensagem digitada lendo a propriedade *Mensagem*, manipulando o valor desta propriedade na variável *Digitado*. Observe que estamos colocando tudo em Maiúsculas (ToUpper) e solicitando que seja impresso o novo valor digitado.

```
using System;

namespace MinhaPrimeiraClasse
{
    class ProgramaPrincipal
    {
        static void Main(string[] args)
        {
            //Criamos o objeto Pedido herdado da clase
InteracaoUsuario
            InteracaoUsuario Pedido = new InteracaoUsu
ario();

            //Invocamos os metodos do objeto Pedido
herdado da classe mãe InteracaoUsuario
            Pedido.SolicitarDigitacao();
```

```
            Pedido.MostrarDigitacao();

            //Recupero pelo metodo GET o valor da
   mensgaem digitada para futura manipulação
            var Digitado = Pedido.Mensagem;
            //Escrevo a nova mensagem colocando toda a
mensagem em Maiuscula
            Console.WriteLine("Nova Digitação sem Espaç
os -> " + Digitado.ToUpper());
        }
    }
    //Criamos a Classe InteracaoUsuario com dois
   metodos principais
    //Metodo que solicita a digitacao de algo no
   terminal
    //Metodo que mostra o que retorna o que foi
   digitado
    class InteracaoUsuario
    {
        private string _MensagemDigitada;
         public string Mensagem
        {
            get
            {
                return _MensagemDigitada;
            }
            set
            {
                _MensagemDigitada = value;
            }
        }
        public void SolicitarDigitacao()
        {
            _MensagemDigitada =  Console.ReadLine();
```

```
        }
        public void MostrarDigitacao( )
        {
            Console.WriteLine(_MensagemDigitada);
        }
    }

}
```

Vamos melhorar o projeto transformando a classe em um arquivo .CS, em uma classe que pode não apenas ser usada no programa, mas em outros programas que necessitem da classe InteracaoUsuario.

Na mesma pasta CAP5 crie uma subpasta OlaMundo_GetSet_Classe e o arquivo Classe_Interacao.cs. Atenção ao nome do namespace MinhaClasseUsuario, o qual será usado no Program.cs.

```
 using System;
 //Classe InteracaoUsuario com dois metodos
principais
 //Metodo que solicita a digitacao de algo no
terminal
 //Metodo que mostra o que foi digitado
 namespace MinhaClasseUsuario
 {
    public class InteracaoUsuario
    {
        private string _MensagemDigitada;
         public string Mensagem
        {
```

```
            get
            { return _MensagemDigitada; }
            set
            { _MensagemDigitada = value; }
        }
        public void SolicitarDigitacao()
        {
            _MensagemDigitada =  Console.ReadLine();
        }
        public void MostrarDigitacao( )
        {
            Console.WriteLine(_MensagemDigitada);
        }
    }
}
```

Em seguida, na mesma pasta OlaMundo_GetSet_Classe crie o arquivo Program.cs. Atente ao *using MinhaClasseUsuario*, que é o *namespace* da classe anterior (InteracaoUsuario), afinal ela será referenciada neste código. É importante usar um *namespace* de fácil assimilação, facilitando o entendimento do programa. Neste caso, o *namespace* desta classe é MinhaPrimeiraClasse, o qual referencia no *using MinhaClasseUsuario* a classe a ser usada neste código.

```
using System;
using MinhaClasseUsuario;

namespace MinhaPrimeiraClasse
{
    class ProgramaPrincipal
    {
```

```
        static void Main(string[] args)
        {
            //Criamos o objeto Pedido herdado da clase
   InteracaoUsuario
            InteracaoUsuario Pedido = new InteracaoUsu
ario();

            //Invocamos os metodos do objeto Pedido
   herdado da classe mãe InteracaoUsuario
            Pedido.SolicitarDigitacao();
            Pedido.MostrarDigitacao();

            //Recuperamos pelo metodo GET o valor da
   mensgaem digitada para futura manipulação
            var Digitado = Pedido.Mensagem;
            //Escrevemos a nova mensagem colocando
   toda a mensagem em Maiuscula
            Console.WriteLine("Nova Digitação sem Espaç
os -> " + Digitado.ToUpper());
        }
    }
}
```

Até aqui conseguimos entender os conceitos básicos de criação da classe e por sua vez a criação de objetos herdados de uma classe mãe que permitirá derivar, organizar, garantir acesso, proteger, estender e reutilizar o código fonte em diversas situações. Mas vamos agora com C# colocar em prática os quatro principais pilares da orientação a objetos que comentamos no começo deste capítulo.

Antes disso vamos entender o que são os construtores de uma

classe, sem dúvida utilíssimos em uma programação orientada a objeto. Até o momento não usamos construtores, isso já é uma evidência de que seu uso não é mandatório nem obrigatório em uma classe. Mas ainda assim vamos estudar seu conceito, sua utilidade e como podemos criar um construtor em C#. Aqui vamos dar início ao principal projeto em C# que irá nortear os principais pilares da orientação a objetos até o final deste capítulo.

Existem vários artigos sobre orientação a objetos, e muitos deles usam carros e seus modelos para explicar OOP, outros usam pessoas e animais para explicar a classe e modelo. Neste código vamos usar um controle de livros em uma biblioteca. Criaremos uma classe que ajudará a controlar nossa biblioteca, nossos livros. No final, veremos particularidades inseridas no C# que também faz parte da OOP e que podem ser muito úteis no dia a dia.

Poderá encontrar este código completo em cada uma de suas versões no GIT chamado “Biblioteca”. Para que haja um melhor acompanhamento de sua leitura, a cada evolução de código, dentro do repositório “Biblioteca” temos as versões de cada evolução “Biblioteca_V1”, “Biblioteca_V2”, e assim por diante.

Dentro da pasta CAP5, crie a subpasta Biblioteca e dentro dela a subpasta “Biblioteca_V1”. Relembrem no programa anterior de interação com o usuário, onde usamos os comandos Console.Writeline e Console.ReadLine. O último passo foi criar uma classe que pudesse estar independente do programa principal e ser utilizada em todo e qualquer projeto. A ideia aqui é igual, o programa de controle de livros na biblioteca será independente.

Adicione o arquivo LibraryClass.cs, conforme o código a seguir.

```
//namespace eque irá conter a classe de controle de livros e seus metodos
namespace Libraryclass
{
    //principal classe de controle de livros na
```

```
Biblioteca
    public class Biblioteca
    {
        private string _Titulo; //recebe o titulo do livro
                          private  string  _Autor;  //
recebe o nome do autor do Livro
        private int _Paginas; ///recebe o númerp de
paginas que o Livro contem
        private bool _Status; //recebe saindo (FALSE) ou
entrando (TRUE) na biblioteca

        public Biblioteca() //Construtor sem parametros
        {

        }
        //Construtor com parametros
         public Biblioteca( string Titulo, string Autor, in
t Paginas, bool Status)
        {
            _Titulo = Titulo;
            _Autor = Autor;
            _Paginas = Paginas;
            _Status = Status;
        }
        public string Titulo //Metodo para acesso a
propriedade Titulo da classe
        {
            get {return _Titulo;} set { _Titulo = value;}
        }
        public string Autor
        {
            get {return _Autor;} set { _Autor = value;}
        }
        public int Paginas
        {
            get {return _Paginas;} set { _Paginas = value;}
```

```
        }
        public bool Status
        {
            get {return _Status;} set { _Status = value;}
        }
    }
}
```

Note que neste código não aplicamos o construtor, na verdade todas as vezes precisarmos de um objeto do tipo *Biblioteca*, teremos que chamar as propriedades GET e SET que são chamadas quando designamos um valor para estas propriedades ou quando solicitamos seu valor atual.

Vamos criar o construtor da Biblioteca usando outro pilar conhecido em orientação a objetos chamado Sobrecarga ou em inglês, *overloading*. Este pilar permite ter dois métodos dentro de uma classe com o mesmo nome, porém com diferentes parâmetros, o que terão diferentes resultados.

Em seguida, adicione o arquivo Program.cs para referenciar a classe *Biblioteca*. Atenção ao *using LibraryClass* declarado para termos acesso à classe *Biblioteca.*

```
using System;
using Libraryclass;
namespace Biblioteca_V1
{
    class Program
    {
        static void Main(string[] args)
        {
            //Inicializando a classe com o construtor nulo
           Biblioteca MinhaBiblioteca = new Biblioteca();
            //Por isso a necessidade de setar o valor de
```

```
cada propriedade
            MinhaBiblioteca.Autor = "Des Dearlove";
              MinhaBiblioteca.Titulo = "O Estilo Bill Gates
de Gerir";
            MinhaBiblioteca.Paginas = 203;
            MinhaBiblioteca.Status = true;
            //imprimindo os valores da classe
              Console.WriteLine("Autor: "+ MinhaBibliotec
a.Autor );
            Console.WriteLine("Titulo: "+MinhaBiblioteca.
Titulo);
             Console.WriteLine("Paginas: "+MinhaBibliote
ca.Paginas);
              Console.WriteLine("Status: "+MinhaBibliotec
a.Status);
            Console.WriteLine();

            //Inicializando a classe com o novo construtor
              Biblioteca MeusLivros = new Biblioteca("Jua
n Garcia Sola", "Linguagem C", 296, true);
            //imprimindo os valores da classe
                 Console.WriteLine("Autor: "+ MeusLivros.
Autor);
          Console.WriteLine("Titulo: "+MeusLivros.Titulo);
                Console.WriteLine("Paginas: "+MeusLivros.
Paginas);
                 Console.WriteLine("Status: "+MeusLivros.
Status);
            Console.WriteLine();
        }
     }
  }
```

Note que classe Biblioteca já possui um construtor da classe, porém não declarado, afinal de contas se este construtor não existisse não conseguiríamos criar o objeto *MinhaBiblioteca*. Quando chamamos o código Biblioteca MinhaBiblioteca = new Biblioteca();

estamos chamando o construtor desta classe e criando um objeto, baseado na classe *Biblioteca* em que a construção desta classe não tem parâmetros. Por isso apenas com a chamada new Biblioteca() invocamos implicitamente o construtor da classe e assim um novo objeto é criado, sem dados, sem autor, sem título, sem nada, uma classe vazia. O objetivo de declarar explicitamente o construtor será útil para no momento de invocarmos a criação de um objeto, já indicarmos quais serão os valores das propriedades deste objeto, sem a necessidade de chamar o código a seguir:

```
MinhaBiblioteca.Autor = “Des Dearlove”;
MinhaBiblioteca.Titulo = “O Estilo Bill Gates de Gerir”;
MinhaBiblioteca.Paginas = 203;
MinhaBiblioteca.Status = true;
```

E como a classe contém dois construtores, temos duas maneiras de inicializar a classe, prova disso é que no Program.cs aparece a sobrecarga de um método. Ao invocar o *new* temos duas maneiras de inicializarmos a classe, com parâmetros e sem parâmetros, conforme a figura a seguir.

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        //Inicializando a classe com o construtor nu
        Biblioteca MinhaBiblioteca = new Biblioteca()
```

```
2/2  Biblioteca.Biblioteca(string Titulo,
     string Autor, int Paginas, bool Status)
```

Figura 21 – Sobrecarga de métodos

Note que no código do construtor com parâmetros não há necessidade de *setar* as propriedades porque elas já são configuradas em tempo de execução do programa, no momento da chamada no construtor.

```
Biblioteca MeusLivros = new Biblioteca(“Juan Garcia Sola”, “Linguagem C”, 296, true);
```

Até aqui nosso programa está indo muito bem. Entendemos so-

bre Classe, método, propriedade, construtor, sobrecarga. Conceitos importantes pra entendermos e programarmos corretamente em C#, já que é uma linguagem nativamente orientada a objetos. Vamos evoluir, vamos falar um pouco agora sobre herança. Herança é um dos atributos fundamentais da programação orientada a objeto. Ela permite que você defina uma classe filha que reutiliza (herda), estende ou modifica o comportamento de uma classe pai. A classe cujos membros são herdados é chamada de classe base. A classe que herda os membros da classe base é chamada de classe derivada.

Dentro da pasta CAP5 e da subpasta Biblioteca, crie a subpasta Biblioteca_V2 e o arquivo Libraryclass.cs. Vamos criar uma classe herdada da classe principal *Biblioteca*. Até aqui a classe base *Biblioteca* está controlando os livros de uma biblioteca. Porém, agora a biblioteca recebe também revistas e precisamos ter o controle do mês e ano da revista, já que geralmente as revistas são mensais. Note que aqui temos duas novas propriedades (mês e ano) que não faz sentido serem incluídas na classe base. Mas, por outro lado as revistas possuem todas as propriedades do livro (autor, páginas, etc) e necessitam que sejam adicionados o mês e o ano. Resolveremos isso com herança.

No arquivo Libraryclass.cs crie uma nova classe chamada *Revistas* que é herdada da classe *Biblioteca*, o qual possui todos os métodos que a revista também necessita. Para que a herança aconteça utilizamos o modificador ":" (dois pontos), sendo:

```
public class Revistas : Biblioteca
```

Pronto, com esta declaração criamos a classe para controlar as revistas, herdando todas as características que possuem a classe mãe, sem que tenhamos que reescrever novamente as propriedades e métodos, pois herdamos todos eles na classe derivada para controlar as revistas. Veja o código completo a seguir.

```
//namespace que irá conter a classe de controle de
```

```
livros e seus metodos
namespace Libraryclass
{
   //nova classe para controlar as revistas
   public class Revistas : Biblioteca
   {
      //Propriedades da Classe
      private int _MesPublicacao;
      private int _AnoPublicacao;
      //Construtor da classe
       public Revistas(int MesPublicacao, int AnoPubli
cacao)
      {
         _MesPublicacao = MesPublicacao;
         _AnoPublicacao = AnoPublicacao;
      }
       public int MesPublicacao //Metodo para acesso
a propriedade da classe
      {
                 get {return _MesPublicacao;} set { _
MesPublicacao = value;}
      }
      public int AnoPublicacao
      {
                 get {return _AnoPublicacao;} set { _
AnoPublicacao = value;}
      }
   }
   //principal classe de controle de livros na
Biblioteca
   public class Biblioteca
   {
      private string _Titulo; //recebra o titulo do livro
      private string _Autor; //recebera o nome do
autor do Livro
      private int _Paginas; ///receberá o númerp de
paginas que o Livro contem
```

```
         private bool _Status; //receberá saindo (FALSE)
ou entrando (TRUE) na biblioteca
        public Biblioteca() //Construtor sem parametros
        {

        }
        //Construtor com parametros
        //note que o nome do metodo é o mesmo
   (sobrecarga)
         public Biblioteca( string Titulo, string Autor, in
t Paginas, bool Status)
        {
           _Titulo = Titulo;
           _Autor = Autor;
           _Paginas = Paginas;
           _Status = Status;
        }
        public string Titulo //Metodo para acesso a
   propriedade da classe "Titulo"
        {
           get {return _Titulo;} set { _Titulo = value;}
        }
        public string Autor
        {
           get {return _Autor;} set { _Autor = value;}
        }
        public int Paginas
        {
           get {return _Paginas;} set { _Paginas = value;}
        }
        public bool Status
        {
           get {return _Status;} set { _Status = value;}
        }
     }
   }
```

Nesta mesma pasta Biblioteca_V2, crie o arquivo Program.cs que irá referenciar e consumir a classe *Revista*.

Note que na chamada do construtor da classe *Revista* são passados os parâmetros mês e ano. Em seguida são inicializadas as outras propriedades que também são importantes para o objeto *Revista* herdado de *Biblioteca*.

Para efeito de deixar o código mais limpo no Console.WriteLine, na segunda linha do using declaramos o *using static System.Console,* fazendo com que usemos apenas o WriteLine.

```
using System;
using static System.Console;
using Libraryclass;
namespace Biblioteca_V2
{
   class Program
   {
      static void Main(string[] args)
      {
         //inicializando nossa nova classe com herança
          Revistas MinhasRevistas = new Revistas(3,20
21);
          //Inicializando as outas propriedades da
classe que foram herdadas
          MinhasRevistas.Autor = “Microsoft”;
          MinhasRevistas.Titulo = “MSDN Magazine”;
          MinhasRevistas.Paginas = 20;
          MinhasRevistas.Status = true;
          //imprimindo os valores de minha classe
            WriteLine(“Revista Mes: “+ MinhasRevistas.
MesPublicacao);
            WriteLine(“Revista Ano: “+ MinhasRevistas.
AnoPublicacao);
           WriteLine(“Revista Autor: “+ MinhasRevistas.
Autor);
```

```
            WriteLine("Revista Titulo: "+MinhasRevistas.
Titulo);
            WriteLine("Revista Paginas: "+MinhasRevista
s.Paginas);
            WriteLine("Revista Status: "+MinhasRevistas.
Status);
            WriteLine();

            //Inicializando nossa classe com o construto
r nulo
            Biblioteca MinhaBiblioteca = new Biblioteca()
;
            //Por isso a necessidade de "setar" o valor
   de cada propriedade
            MinhaBiblioteca.Autor = "Des Dearlove";
            MinhaBiblioteca.Titulo = "O Estilo Bill Gates
de Gerir";
            MinhaBiblioteca.Paginas = 203;
            MinhaBiblioteca.Status = true;
            //imprimindo os valores de minha classe
            WriteLine("Autor: "+ MinhaBiblioteca.Autor);
            WriteLine("Titulo: "+MinhaBiblioteca.Titulo);
          WriteLine("Paginas: "+MinhaBiblioteca.Paginas);
            WriteLine("Status: "+MinhaBiblioteca.Status);
            WriteLine();

            //Inicializando nossa classe com o novo
   construtor
            Biblioteca MeusLivros = new Biblioteca("Jua
n Garcia Sola", "Linguagem C", 296, true);
            //imprimindo os valores de minha classe
            WriteLine("Autor: "+ MeusLivros.Autor);
            WriteLine("Titulo: "+MeusLivros.Titulo);
            WriteLine("Paginas: "+MeusLivros.Paginas);
            WriteLine("Status: "+MeusLivros.Status);
            WriteLine();
        }
```

```
    }
}
```

Salve o projeto (CTRL+S) e execute (F5) para ver o resultado.

```
Autor: Des Dearlove
Titulo: O Estilo Bill Gates de Gerir
Paginas: 203
Status: True
Autor: Linguagem C
Titulo: Juan Garcia Sola
Paginas: 296
Status: True
```

Vamos abordar agora o polimorfismo, outro pilar da orientação a objetos e que nos ajudará a codificar melhor, mais rápido e reutilizando código. O principal conceito do polimorfismo é a propriedade de duas ou mais classes derivadas de uma mesma superclasse responderem a mesma mensagem cada uma de uma forma diferente, mas utilizam o mesmo método que outrora foi sobrescrito.

Vamos ver isso na prática, vamos complicar um pouco mais nosso controle na biblioteca. Agora, além de controlar livros e revistas precisamos controlar Fotos, com o agravante que jornal tem todas as características de um livro, possui (Autor, Título, etc), porém um novo campo que chamaremos de *Tipo,* o qual armazenará o valor Colorido ou Preto e Branco.

Já aprendemos neste capítulo como resolver isso, basta criarmos a classe *Fotos* herdada da classe *Revistas,* assim teremos todas as propriedades à disposição para trabalhar sem precisar reescrever. Vamos fazer isso antes de aplicarmos polimorfismo.

```
//classe para controlar as fotos
public class Fotos : Revistas
{
    //Propriedades da Classe
```

```
    private string _Tipo;
    //neste construtor preciso referenciar a classe
base
    //Revistas é uma classe já herdada de Biblioteca
    public Fotos(string Tipo) : base (0,0)
    {
        _Tipo = Tipo;
    }
    public string Tipo //Metodo para acesso a
propriedade da classe
    {
        get {return _Tipo;} set { _Tipo = value;}
    }
}
```

Para consumir a classe *Fotos*, no Program.cs adicione o código a seguir. Criamos o objeto *MinhasFotos* derivada da classe *Fotos* e designo valores para suas propriedades. Note que a classe *MinhasFotos* possui todas as propriedades e métodos que foram herdados da classe *Revistas* que por sua vez herdou da classe *Biblioteca*.

```
static void Main(string[] args)
{
    //inicializando nossa nova classe Fotos
    Fotos MinhasFotos = new Fotos("Colorida");
    MinhasFotos.AnoPublicacao = 2021;
    MinhasFotos.MesPublicacao = 3;
    MinhasFotos.Autor = "Desconhecido";
    MinhasFotos.Titulo = "Vida Eterna";
    MinhasFotos.Paginas = 1;
    MinhasFotos.Status = true;
    //imprimindo os valores de minha classe
    WriteLine("Foto Tipo: "+ MinhasFotos.Tipo);
  WriteLine("Foto Mes: "+ MinhasFotos.MesPublicacao);
  WriteLine("Foto Ano: "+ MinhasFotos.AnoPublicacao);
    WriteLine("Foto Autor: "+ MinhasFotos.Autor);
    WriteLine("Foto Titulo: "+MinhasFotos.Titulo);
```

```
    WriteLine("Foto Paginas: "+MinhasFotos.Paginas);
    WriteLine("Foto Status: "+MinhasFotos.Status);
    WriteLine();
```

Agora entra a utilidade do polimorfismo, ou seja, a classe obtém múltiplas formas e múltiplos comportamentos. Vamos agora, dentro de nosso exercício proposto, criar um método de descarte do Livro, da revista e da Foto. Todos terão o mesmo objetivo de descartar, porém para cada um o descarte é realizado de maneira diferente, o jornal é reciclado, o livro é doado e a foto é destruída. Para solucionar este problema podemos criar um método base na classe base chamado *Descarte* e designar diferentes comportamentos nas classes derivadas.

Veja os métodos Descarte nas classes *Biblioteca*, *Revistas* e *Fotos*. Note que a propriedade Titulo de cada classe tem um texto diferente. Nas classes derivadas *Revistas* e *Fotos,* a declaração da propriedade *Titulo*, usamos o *this,* o qual indica que é desta classe.

```
namespace Libraryclass
{
   //classe para controlar as fotos
   public class Fotos : Revistas
   {
      //Propriedades da Classe
      private string _Tipo;
      ...
      public string Tipo //Metodo para acesso a
propriedade daclasse      {
         get {return _Tipo;} set { _Tipo = value;}
      }
      public override void Descarte()
      {
         this.Titulo = "A Foto foi Destruida";
      }
   }
```

```
    //nova classe para controlar as revistas
    public class Revistas : Biblioteca
    {
        //Propriedades da Classe
        private int _MesPublicacao;
        private int _AnoPublicacao;
        ...
        public int AnoPublicacao
        {
                    get {return _AnoPublicacao;} set { _
AnoPublicacao = value;}
        }
        public override void Descarte()
        {
            this.Titulo = “A Revista foi Reciclada”;
        }
    }
    //principal classe de controle de livros na
  Biblioteca
    public class Biblioteca
    {
        private string _Titulo; //recebra o titulo do livro
        ...
        ...
        public bool Status
        {
            get {return _Status;} set { _Status = value;}
        }

        //Método Descarte
        public virtual void Descarte()
```

```
        {
            _Titulo = “O Livro foi Doado”;
        }


    }
}
```

No Program.cs, adicione este bloco de código no final, assim teremos os objetos de foto, jornal e o livro na biblioteca com os respectivos descartes.

```
//Mesmo método com comportamentos diferentes
MeusLivros.Descarte();
WriteLine(“Titulo: “+MeusLivros.Titulo);
MinhasRevistas.Descarte();
WriteLine(“Titulo: “+MinhasRevistas.Titulo);
MinhasFotos.Descarte();
WriteLine(“Titulo: “+MinhasFotos.Titulo);
WriteLine();
```

Este é o resultado esperado ao rodar o programa.

Salve o projeto (CTRL+S) e execute (F5) para ver o resultado.

```
Foto Tipo: Colorida
Foto Mes: 3
Foto Ano: 2021
Foto Autor: Desconhecido
Foto Titulo: Vida Eterna
Foto Paginas: 1
Foto Status: True
```

```
Revista Mes: 3
Revista Ano: 2021
Revista Autor: Microsoft
Revista Titulo: MSDN Magazine
Revista Paginas: 20
Revista Status: True

Autor: Des Dearlove
Titulo: O Estilo Bill Gates de Gerir
Paginas: 203
Status: True

Autor: Linguagem C
Titulo: Juan Garcia Sola
Paginas: 296
Status: True

Titulo: O Livro foi Doado
Titulo: A Revista foi Reciclada
Titulo: A Foto foi Destruida
```

# CAPÍTULO 6 – TRATAMENTO DE EXCEÇÕES

Uma exceção é um erro causado por alguma operação dentro do software, como por exemplo: falha em ler um arquivo do disco, falha em se conectar a um banco de dados, uma variável sem um valor adequado, entre muitos outros. O importante neste caso é tratarmos o erro de maneira que o usuário da aplicação receba alguma informação que faça sentido para ele.

Outro aspecto no tratamento de exceções é relativo à segurança da aplicação, pois se deixarmos um erro aparecer "em sua forma bruta" para o usuário, podemos estar expondo dados sensíveis da nossa aplicação, como banco de dados que usamos, tecnologia, linguagem, além de outros dados que podem ser utilizados sem o nosso conhecimento.

Para entendermos os erros e como trata-los de maneira correta, iremos explorar o mecanismo de tratamento de exceções da linguagem C#.

## O QUE É UM ERRO?

Um erro é um comportamento que pode paralisar nossa aplicação, seu fluxo normal de execução, então, quando um erro acontece, podemos chama-lo de "exceção", pois é algo que não planejamos, e claro, não planejamos erros em nossa aplicação, mas precisamos planejar o que fazer com eles caso aconteçam.

Alguns exemplos bem comuns de erros:

- Banco de dados indisponível: por algum erro de rede ou alto nível de processamento, nossa aplicação não consegue acessar o banco de dados.
- Dados Nulos: chamamos uma função externa e o valor retornado é um nulo e não o que deveria voltar de fato.
- Erros Matemáticos: um erro bem comum nesta área é o erro de divisão por zero.

E assim poderíamos listar aqui centenas de tipos de erros que podem acontecer em uma aplicação.

## SEMPRE TEREMOS ERRO EM NOSSA APLICAÇÃO?

Nós não criamos uma aplicação para ela conter erros, mas eles podem acontecer, e como um bom desenvolvedor, precisamos tratar estes erros, evitando problemas maiores na aplicação. Além disto, existem recursos que consumimos em uma aplicação que estão fora o nosso controle, por exemplo: dados acessados pela internet, um serviço de banco de dados externo, uma API, ou seja, temos muitos pontos em uma aplicação que realmente podem falhar e nossa missão é tratar tudo isto da melhor maneira possível.

## COMO TRATAMOS UM ERRO?

Já entendemos que erros podem acontecer e que precisamos tratá-los, o que precisamos agora é saber a melhor maneira de fazer isto e principalmente, identificar pontos de falha em nosso código que possam gerar um problema.

Para iniciarmos com o tratamento de erros, vejamos o código a seguir, que é bem simples e trata da divisão de dois números informados pelo usuário:

```
using System;

namespace Cap06
{
    public class TratamentoExcecao
    {
        public void ExemploTratamentoExcecao()
        {
            Console.WriteLine("Divisão de dois números");
            Console.WriteLine("");
            Console.WriteLine("Digite o primeiro número :");
            var strNum1 = Console.ReadLine();
            Console.WriteLine("Digite o segundo número :");
            var strNum2 = Console.ReadLine();

            int num1 = int.Parse(strNum1);
            int num2 = int.Parse(strNum2);
            var divisao = num1 / num2;
            Console.WriteLine($"A divisão de {num1} por {num2} é {divisao}");
            Console.WriteLine("Fim");
        }
    }
}
```

Vamos entender o código: pegamos o primeiro e o segundo número e guardamos em duas variáveis chamadas: *num1* e *num2*. Como a função *Console.ReadKey()* retorna apenas uma string, precisamos converter *num1* e *num2* para int, então usamos a função *int. Parse()* que converte uma string para um int. Finalmente pegamos os dois valores agora como inteiros e fazemos a divisão.

Primeiro vamos fazer o que chamamos de "caminho feliz", ou seja, vamos usar a nossa aplicação como ela foi criada, informando os valores corretamente:

```
Divisão de dois números

Digite o primeiro número: 10
Digite o segundo número: 2
A divisão de 10 por 10 é 1
Fim
```

A pergunta aqui é o que poderia dar de errado neste código tão simples? E a resposta é: muitas coisas, vamos lá: o usuário não digita nada no campo dos números, o que provoca um erro na conversão para int. Isto resulta nesta mensagem de erro:

```
Divisão de dois números

Digite o primeiro número:

Digite o segundo número:

Unhandled exception. System.FormatException: Input string was not in a correct format.
   at System.Number.ThrowOverflowOrFormatException(ParsingStatus status, TypeCode type)
   at System.Number.ParseInt32(ReadOnlySpan`1 value, NumberStyles styles, NumberFormatInfo info)
   at System.Int32.Parse(String s)
   at livrocsharp.Program.Main(String[] args) in D:\cds\git\livros\livrocsharp\Cap6\Program.cs:line 16

Agora o usuário digita um valor inválido no campo:

Divisão de dois números

Digite o primeiro número:
0
Digite o segundo número:
0
Unhandled exception. System.
```

```
DivideByZeroException: Attempted to divide by zero.
    at livrocsharp.Program.Main(String[] args) in D:\
cds\git\livros\livrocsharp\Cap6\Program.cs:line 18
```

Veja que em nenhum dos dois exemplos, a palavra "Fim" foi impressa na tela, pois ela estava depois da conta de dividir, onde o erro foi provocado. Então o erro interrompeu o fluxo da nossa aplicação.

O que precisamos fazer é tratar o erro e seguir com a execução da aplicação, então vamos incluir no nosso exemplo um tratamento de exceção, veja agora como fica:

```
using System;

namespace Cap06
{
    public class TratamentoExcecao
    {
        public void ExemploTratamentoExcecao()
        {
            Console.WriteLine("Divisão de dois números");
            Console.WriteLine("");
            Console.WriteLine("Digite o primeiro número
:");
            var strNum1 = Console.ReadLine();
            Console.WriteLine("Digite o segundo número
:");
            var strNum2 = Console.ReadLine();

            try
            {
                int num1 = int.Parse(strNum1);
                int num2 = int.Parse(strNum2);
                var divisao = num1 / num2;
                Console.WriteLine($"A divisão de {num1} p
or {num2} é {divisao}");
```

```
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Erro na divisão: {ex.
Message}");
            }
            Console.WriteLine("Fim");
        }
    }
}
```

Quem resolve o problema do tratamento de exceção no C# é o bloco *try..catch..finaly*. Este bloco é o responsável por tratar qualquer exceção que ocorra dentro do *try*, pegando o erro no *catch*, e em alguns casos, executando códigos no *finally*.

Então o código que queremos proteger fica dentro do bloco *try*, que no nosso exemplo é a conversão do dado e a divisão. Se acontecer algum erro dentro do try, então o bloco *catch* é chamado, e no nosso caso, mostramos uma mensagem que recebemos de erro. O nosso exemplo não tem um *finally*, mas iremos explorar este bloco logo mais.

Vamos agora executar os mesmos dois exemplos que provocaram erros:

```
Divisão de dois números

Digite o primeiro número:

Digite o segundo número:

Erro na divisão: Input string was not in a correct
format.
Fim
```

E agora com divisão por zero:

```
Divisão de dois números

Digite o primeiro número:
0
Digite o segundo número:
0
Erro na divisão: Attempted to divide by zero.
Fim
```

Veja que agora temos uma mensagem de erro mais simples, e também temos a palavra “Fim” sendo mostrada nos dois casos.

O bloco try..catch..finally é muito útil mas não deve ser utilizado em toda a sua aplicação, ou seja, não comece o código com um try, coloque o bloco onde você realmente precisa. E isto você vai aprendendo com o uso da linguagem e a construção de aplicações. Mas te pedimos para nunca escrever um bloco try como do exemplo a seguir:

```
public static void NaoFacaTryCatchAssim()
{
    try
    {
      int a = 1;
      int b = 0;
      int c = a/b;
    }
    catch
    {
    }
}
```

Este código provoca um erro que pode nunca ser encontrado, pois o *catch* não mostra nada, absolutamente nada. Se você não tem certeza do que escrever no catch, use o comando *throw*:

```
public static void NaoFacaTryCatchAssim()
{
    try
    {
      int a = 1;
      int b = 0;
      int c = a/b;
     }
     catch
     {
      throw;
     }
}
```

O *throw* joga a exceção para quem chamou o código.

### TRATANDO ERROS ESPECÍFICOS COM TRY..CATCH

Vimos como tratar um erro dentro do nosso código C#, mas até agora estamos tratando este erro de maneira genérica, através da classe *Exception*, mas o C# tem uma hierarquia de tratamento de erros, que vai do mais específico para o mais genérico.

Isto pode ter parecido um pouco estranho, então vamos explicar usando o código do nosso exemplo anterior, mas incrementando o tratamento de erros:

```
using System;

namespace Cap06
{
    public class TratamentoExcecao
    {
        public void ExemploTratamentoExcecao()
        {
            Console.WriteLine("Divisão de dois números");
             Console.WriteLine("");
```

```
            Console.WriteLine("Digite o primeiro número
:");
            var strNum1 = Console.ReadLine();
            Console.WriteLine("Digite o segundo número
:");
            var strNum2 = Console.ReadLine();

            try
            {
                int num1 = int.Parse(strNum1);
                int num2 = int.Parse(strNum2);
                var divisao = num1 / num2;
                Console.WriteLine($"A divisão de {num1} p
or {num2} é {divisao}");
            }
            catch (DivideByZeroException ex)
            {
                Console.WriteLine($"Erro de Divisão por Ze
ro: {ex.Message}");
            }
            catch (FormatException ex)
            {
                Console.WriteLine($"Erro de Formatação: {
ex.Message}");
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Erro: {ex.Message}");
            }
            Console.WriteLine("Fim");
        }
    }
}
```

Veja que agora temos três blocos catch, onde o primeiro trata erro de divisão por zero, o segundo trata erro de formato e o último trata qualquer erro que não foi tratado pelos anteriores. Então é

bem simples, vamos tratar os erros do mais específico para o mais genérico.

Mas qual a vantagem disto? A vantagem é que usando uma classe específica podemos pegar mais detalhes dos erros.

Se executarmos novamente o código usando os mesmos dados dos exemplos anteriores teremos o resultado a seguir:

```
Divisão de dois números

Digite o primeiro número:

Digite o segundo número:

Erro de Formatação: Input string was not in a correct format.
Fim
```

Temos a mensagem "Erro de Formatação" pois o erro foi provocado por um dado inserido incorretamente e que não pode ser convertido.

```
Divisão de dois números

Digite o primeiro número:10
Digite o segundo número:0
Erro de Divisão por Zero: Attempted to divide by zero.
Fim
```

E agora temos uma mensagem específica de divisão por Zero.

## O COMANDO FINALLY

Agora que já entedemos o *try..catch*, vou adicionar o segmento

*finally*, que sempre é executado, independente do que ocorra no *try* ou no *catch*. Vamos ver no código a seguir:

```
using System;

namespace Cap06
{
    public class TratamentoExcecao
    {
        public void ExemploTratamentoExcecao()
        {
            Console.WriteLine("Divisão de dois números");
             Console.WriteLine("");
              Console.WriteLine("Digite o primeiro número
:");
             var strNum1 = Console.ReadLine();
              Console.WriteLine("Digite o segundo número
:");
             var strNum2 = Console.ReadLine();

             try
             {
                 int num1 = int.Parse(strNum1);
                  int num2 = int.Parse(strNum2);
                  var divisao = num1 / num2;
                  Console.WriteLine($"A divisão de {num1} p
or {num2} é {divisao}");
             }
             catch (DivideByZeroException ex)
             {
                 Console.WriteLine($"Erro de Divisão por Ze
ro: {ex.Message}");
             }
             catch (FormatException ex)
             {
                  Console.WriteLine($"Erro de Formatação: {
ex.Message}");
```

```
            }
            catch (Exception ex)
            {
               Console.WriteLine($"Erro: {ex.Message}");
            }
            finally
            {
                Console.WriteLine("Sempre vai executar o
Finally");
            }
            Console.WriteLine("Fim");
        }
    }
}
```

E o resultado é este:

```
Divisão de dois números

Digite o primeiro número:
10
Digite o segundo número:
2
A divisão de 10 por 2 é 5
Sempre vai executar o Finally
Fim
```

O *finally* apareceu antes do "Fim" pois no código ele está realmente antes.

O tratamento de erros, de maneira correta, é extremamente importante para uma aplicação. Lembre-se de não deixar estourar erros sem sentido na tela do seu usuário, sempre mostrando informações que fazem sentido e orientem para o correto uso da aplicação.

# CAPÍTULO 7 – MÉTODO DE EXTENSÃO, DICIONÁRIOS, PARÂMETROS OPCIONAIS E DELEGATES COM FUNC<>

O C# permite ao desenvolvedor criar códigos próprios para auxiliar na produtividade e uso em qualquer parte do projeto, por exemplo, um método de extensão para auxiliar na formatação de um número, você cria uma vez um determinado formato customizado e o usa em todo o projeto, apenas chamando o método.

Já em casos de percorrer uma lista de dados, não importa qual o tipo, usamos os loopings (For, ForEach, While), a fim executar o processamento do mesmo conjunto de códigos até o fim do ciclo.

E existem outros meios de definir uma estrutura de dados, os chamados dicionários de dados e fazer a chamada de métodos customizados ou não, passando parâmetros como referência. No entanto, veremos como declarar parâmetros opcionais, passando ou não o dado, o método é executado com valores padrão ou não.

Neste capítulo, vamos aprender passo a passo os tópicos citados acima, pois são usados praticamente em todos os projetos porque são temas usuais.

Para isto, no VS Code, crie uma nova pasta chamada Cap7 onde teremos todos os exercícios C#. Você pode clicar no segundo ícone, conforme a figura a seguir ou clicar com o botão direito na lista de itens da janela do *Explorer* e selecionar *New Folder*.

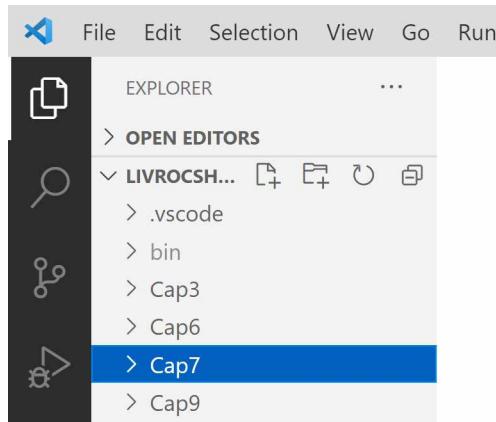



Figura 22 - Nova pasta CAP7

Com isto, todos os exercícios deste capítulo deverão ser adicionados à esta pasta Cap7.

## MÉTODO DE EXTENSÃO

O conceito de método de extensão é criar um método customizado para um determinado tipo de dado (*int, DateTime, string*, etc) e aplicar uma ação. Os tipos primitivos do C# contém todas as possibilidades possíveis de customização, no entanto, de acordo com o tipo de aplicação é normal o desenvolvedor criar seus próprios formatos e funções, a fim de facilitar o uso e ter produtividade.

Vamos a um exemplo. Adicione um novo arquivo chamado MetodoExtensao.cs à pasta Cap7. Em seguida, digite a lista de *USING*, o *namespace*, o nome da classe e o *SVM* (*static void Main*).

Para um método de extensão é regra obrigatória que a classe (*public static class*) e os métodos (*public static string*) sejam estáticos. Digite o método *FormatarData*, note que na declaração ele retorna um texto (*string*), recebe como entrada um campo do tipo data *DateTime* e outro do tipo texto (*string*). Este método receberá uma data e aplicará o formato de acordo com o que o desenvolvedor

informar na variável formato.

Note ainda que a variável data está declarada com o *THIS*, o que significa que pertence a esta classe, ou seja, é dela mesma. E qual o retorno? O retorno do método está após a lambda => no código *data.ToString(formato).* Então, a esta data será aplicado através do método *ToString*() do C#, o formato que foi passado pelo desenvolvedor na variável *formato*.

Em seguida, digite o método *Formatar*, que também está declarado como *static*, retorna uma *string*, recebe o número do tipo *decimal* (apenas *decimal* e não *int* ou *double*), aplica o formato pelo *ToString*() de acordo com o formato passado ao método. Isto significa que ora pode ser um formato monetário, somente número, com zero ou 2 casas decimais, etc.

```
using System;
using static System.Console;

namespace livrocsharp
{
    public static class MetodosExtensao
    {

        // o método de extensão DEVE ser static
        public static string FormatarData(
            this DateTime data, string formato)
            => data.ToString(formato);

        public static string Formatar(
            this decimal valor, string formato)
            => valor.ToString(formato);
```

```
    }
}
```

Agora com os dois métodos declarados, basta usarmos. Dentro desta classe *MetodosExtensao* declare o *SVM* (*static void Main*) e digite os códigos a seguir. Lembre-se que para uso do método *FormatarData* o dado DEVE ser do tipo *DateTime*, assim como para o método *Formatar*, o dado DEVE ser *decimal*.

O primeiro bloco para exibir a data formatada, usamos o dia atual *DateTime*.*Today*. E, para os devidos formatos de dia, mês, ano, dia da semana, basta informar entre aspas o formato a ser exibido, afinal é o segundo parâmetro do método de extensão.

Para o número decimal, declare a variável *valor*, chame o método *Formatar* e aplique o formato monetário customizado “R$” ou apenas o “C2”, que é de *Currency* com 2 casas decimais. Veja que no exemplo, há um número do tipo *decimal* digitado diretamente na linha (4578.78M).

```
public static class MetodosExtensao
{
    static void Main(string[] args)
    {
        WriteLine(DateTime.Today.FormatarData(“dd/MMM/yyyy”));
        WriteLine(DateTime.Today.FormatarData(“dddd dd/MMM/yyyy”));
        WriteLine(DateTime.Today.FormatarData(“MMMM yyyy”));
        WriteLine(DateTime.Today.FormatarData(“yyyy”));

        decimal valor = 5800.78M;
```

```
        WriteLine(valor.Formatar("R$ ##,##0.00"));

        WriteLine($"{4578.87M.Formatar("C2")}");
        WriteLine(4578.87M.Formatar("C2"));
        ReadLine();
    }
```

O bom do VS Code é que durante a codificação, quando for aplicar um método de extensão, ele já aparece no *Intellisense*, fazendo parte da lista de métodos.

```
static void Main(string[] args)
{
    WriteLine(DateTime.Today.forma)
                             FormatarData
    decimal valor =          TryFormat
    WriteLine(valor.         GetDateTimeFormats
```

Figura 23 – Intellisense no VS Code

Salve o arquivo, declare-o no *StartupObject* do livrocsharp.csproj e pressione F5 para executa-lo.

```
<StartupObject>livrocsharp.MetodosExtensao</StartupObject>
```

***Resultado:***

05/jul/2021
segunda-feira 05/jul/2021
julho 2021
2021
R$ 5.800,78

```
R$ 4.578,87
```

Agora que temos estes métodos declarados, em qualquer parte do projeto você pode usufruir deste recurso, basta invocar o método de extensão. É comum nos times de desenvolvimento criar uma biblioteca de utilitários de formatações, assim quando precisar alterar um determinado formato em toda a aplicação, isto é feito apenas uma única vez direto no método de extensão, o qual será replicado em todos os lugares que o usa.

## PARÂMETROS OPCIONAIS

Em qualquer linguagem de programação há muitas funções, métodos com diversas quantidades de parâmetros a serem passados. Muitas vezes somos obrigados a passar parâmetros para um método, mesmo que estes não sejam usados para nada, apenas por convenção da escrita de código de quem o criou. Nas primeiras versões do C# e até hoje podemos criar o que chamamos de sobrecarga de método, ou seja, a assinatura do método é a mesma, só que a quantidade de parâmetro é diferente. Por isto que em diversos métodos temos a indicação de 1 de 7, 2 de 7, ou seja, temos 7 opções da chamada.

Sendo assim, foi adicionado ao C# o chamado parâmetro opcional, ou seja, você pode ou não informar o valor. Caso não seja informado, o C# assume um valor padrão.

Adicione um novo arquivo chamado *ParametrosOpcionais*.cs à pasta Cap7. Em seguida, digite a lista de *USING*, o *namespace*, o nome da classe e o *SVM (static void Main*).

Note que o método Faturamento recebe 3 parâmetros, sendo *qtde, preco e bonus*. O primeiro (*qtde*) é obrigatório, já os demais são opcionais. O *preco* assume o valor de 10, caso não seja informado o valor, assim como o *bonus* assume o valor de 5.

Existe uma regra para esta declaração: Todos os parâmetros

obrigatórios DEVERÃO ser declarados ANTES dos opcionais. Então, respeite isto que jamais terá problema.

Sendo assim, o método Faturamento retorna um número do tipo *decimal* com o resultado da fórmula que é atribuído o cálculo => *qtde* * *preco* + *bonus*. O que notamos de diferente nos parâmetros são os valores atribuídos ao *preco* = 10 e *bonus* = 5.

```
using System;
using static System.Console;

namespace livrocsharp
{
   class ParametrosOpcionais
   {
      // regra: obrigatórios DEVEM vir antes dos opcionais
      static decimal Faturamento(
            decimal qtde, decimal preco = 10, decimal bonus = 5)
            => qtde * preco + bonus;

      static void Main(string[] args)
      {
         WriteLine("Parâmetros Opcionais");
         WriteLine(Faturamento(500, 5, 10));
         WriteLine("Parâmetros 1 e 2");
         WriteLine(Faturamento(50, 12));
         WriteLine("Parâmetros 1 e 3");
         WriteLine(Faturamento(50, bonus: 10));
         ReadLine();
      }
   }
}
```

Para testar o resultado, digite as linhas do *WriteLine* para mostrar os resultados, sendo que o primeiro número (*qtde*) está informado

em todas as 3 chamadas do Faturamento. No primeiro *WriteLine* passamos os 3 valores. No segundo *WriteLine* passamos a *qtde* (50) e o *preco* (12) e não passamos o *bonus*, o qual assumirá o valor de 5.

Já no terceiro *WriteLine* passamos a *qtde* (50) e o *bonus* (10) e não passamos o segundo parâmetro que é o *preco*, assumindo assim o valor padrão de 10. Atente a sintaxe usada, declaramos o nome do parâmetro (*bonus*), seguido de dois pontos ( : ) com o respectivo valor. Ou seja, quando tiver uma lista de parâmetros e deseja referenciar um opcional, basta digitar o nome e o valor.

Salve o arquivo, declare-o no *StartupObject* do livrocsharp.csproj e pressione F5 para executa-lo.

```
<StartupObject>livrocsharp.ParametrosOpcionais</StartupObject>
```

***Resultado:***

**Parâmetros Opcionais**
**2510**
**Parâmetros 1 e 2**
**605**
**Parâmetros 1 e 3**
**510**

## DICIONÁRIO

Se você precisar pesquisar o significado de qualquer palavra recorre ao dicionário. Este armazenamento se dá através de uma pesquisa pela palavra (chave), retornando os significados (resultado, valor). Isto também se aplica a uma linguagem de programação, e no C# temos este recurso através da seguinte declaração:

```
public Dictionary(IDictionary<TKey, TValue> dictionary);
```

Vamos a um exemplo prático, adicione um novo arquivo Dicionario.cs à pasta Cap7. Em seguida, digite a lista de *USING*, o *namespace*, o nome da classe e o *SVM (static void Main*) onde teremos todos os exemplos.

```
using System;
using System.Collections.Generic;
using System.Linq;
using static System.Console;

namespace livrocsharp
{
    class Dicionario
    {
        static void Main(string[] args)
        {

        }
    }
}
```

Dentro do *Main*() digite este bloco de código onde declaramos a variável sexo como um *Dictionary*, sendo que a chave e o valor são do tipo *string*, ou seja, teremos textos na chave e no conteúdo.

Em seguida, para adicionar um item ao dicionário, usamos o método Add, seguido dos textos da chave e valor. Neste caso, temos a chave M para Masculino e F para Feminino.

Quando precisar ter acesso ao conteúdo da chave, basta referenciar o texto da chave, neste caso F. A sintaxe requer o nome do dicionário com a chave, sendo *sexo[“F”]*.

```
WriteLine(“--- uso de dicionarios ---”);
var sexo = new Dictionary<string, string>();
sexo.Add(“M”, “Masculino”);
```

```
sexo.Add("F", "Feminino");
WriteLine(sexo["F"]);
```

Salve o arquivo, declare-o no *StartupObject* do livrocsharp.csproj e pressione F5 para executa-lo.

```
<StartupObject>livrocsharp.ParametrosOpcionais</StartupObject>
```

***Resultado:***

**--- uso de dicionarios ---**
**Feminino**

O próximo exemplo usa uma sintaxe diferente, não é preciso usar o método *Add*, basta informar os respectivos dados (chave e valor) separados por vírgula. Neste caso, temos uma variável chamada *cidades* do tipo *Dictionary*, cuja chave e valor são do tipo texto (*string*). Em seguida, são adicionados os conteúdos da lista.

```
var cidades = new Dictionary<string, string>
{
    ["MG"] = "Minas Gerais",
    ["SC"] = "Santa Catarina",
    ["SP"] = "São Paulo",
    ["RS"] = "Rio Grande do Sul"
};
WriteLine($"--- dicionário com {cidades.Count()} Cidades ---");
WriteLine(cidades["MG"]);
//WriteLine(cidades["mg"]); // gera erro
//WriteLine(cidades[0]); // erro compilação
foreach (var cidade in cidades)
{
    WriteLine($"sigla: {cidade.Key} - {cidade.Value}");
}
```

Note que para saber a quantidade de elementos no dicionário usamos o *Count*(), ou seja, basta contar o número de elementos.

Para exibir o conteúdo (nome do estado) é preciso buscar pela chave, que neste caso é o "MG". No entanto, se você buscar por "mg" dará um erro em tempo de execução, pois não existe. Todas as chaves são sensíveis a escrita, maiúsculas são diferentes de minúsculas. Outro tipo de erro que ocorre, neste caso, erro de compilação é o uso pelo índice [0]. Isto não é permitido neste caso, pois a chave é alfanumérica.

Em seguida, para listar todo o dicionário, usamos um *looping* do tipo *foreach* varrendo toda a lista. Note que a sintaxe do *foreach* temos a declaração do tipo *var* cidade (nome qualquer de uma variável criada em tempo de compilação), seguido da variável que contém todo o dicionário, neste caso, *in cidades.*

Para listar a chave e o conteúdo de cada item do dicionário dentro do *looping foreach,* usamos o *WriteLine* para ler a chave (*cidade.Key*) e o valor (*cidade.Value*). Sendo assim, qualquer acesso à chave ocorre pela palavra chave *Key* e para o valor, o *Value*.

O uso do *looping foreach* varre todos os itens da coleção *cidades*, ou seja, você não precisa se preocupar quando terminará a leitura.

Salve e execute F5 para ver o resultado.

***Resultado:***

**--- dicionário com 4 Cidades ---**
**Minas Gerais**
**sigla: MG - Minas Gerais**
**sigla: SC - Santa Catarina**
**sigla: SP - São Paulo**
**sigla: RS - Rio Grande do Sul**

Neste próximo exemplo, teremos a chave como número e o con-

teúdo como *string*. Veja como que a variável países é definida, chave como *int* e conteúdo/valor como *string*.

```
WriteLine("--- dicionário Países ---");
var paises = new Dictionary<int, string>
{
    [100] = "Brasil",
    [2] = "Australia",
    [60] = "Nova Zelandia"
};
foreach (var pais in paises)
{
    WriteLine($"{pais.Key} - {pais.Value}");
}
```

***Resultado:***

**--- dicionário Países ---**
**100 - Brasil**
**2 - Australia**
**60 - Nova Zelandia**

Neste exemplo, usaremos uma coleção de itens num array de string. Veja a declaração da variável texto com 8 palavras separadas, sendo que cada uma com um índice (no C# o índice começa em zero) de início, assim como o índice do final, representado pelo símbolo circunflexo (^) + número.

```
var texto = new string[]
{
                   // índice do início    índice do final
    "Meu",            // 0                   ^9
    "codigo",         // 1                   ^8
    "rodou",          // 2                   ^7
    "sem",            // 3                   ^6
```

```
    "erros",        // 4                    ^5
    "para",         // 5                    ^4
    "ler",          // 6                    ^3
    "cadastro",     // 7                    ^2
    "clientes"      // 8                    ^1
};
WriteLine("--- Índices do dicionário ---");
```

Com os índices associados às palavras, como saber qual é a primeira e a última palavra do *array*? Basta referenciar o ^9 para a primeira e o ^1 para a última. Note que esta sintaxe não usamos o ^0, e sim o ^1 para o final da lista.

```
WriteLine($"A última palavra é {texto[^1]}");
WriteLine($"A primeira palavra é : {texto[^9]}");
```

E para capturar apenas uma parte dos itens do *array*? Veja a declaração da variável *pedacoTexto* que captura as palavras contidas entre os índices 1 e 4, inclusive. No *looping foreach* é usada a coleção pedacoTexto, contendo só as palavras de 1 a 4.

```
WriteLine("--- parte do texto ---");
var pedacoTexto = texto[1..4];
foreach (var p in pedacoTexto)
{
    WriteLine(p);
}
```

Uma opção é usar o *Range* de um intervalo, neste caso, a variável intervalo contém os índices de 1 a 6. E no *looping foreach* é usado a variável texto seguido do número do índice do intervalo, por exemplo, texto[1], texto[2], etc.

```
WriteLine("--- intervalo da coleção ---");
```

```
Range intervalo = 1..6;
foreach (var p in texto[intervalo])
{
   WriteLine(p);
}
```

Caso queira capturar todas as palavras use apenas o .. na lista de índice, texto[..]. Para ler apenas as 4 primeiras palavras, use texto[..4].

```
var todas = texto[..];   // todas as palavras
WriteLine("--- primeiras 4 ---");
var Primeiras4 = texto[..4]; // primeiras 4 palavras
foreach (var p in Primeiras4)
{
   WriteLine(p);
}
```

Para ler as 4 últimas palavras, use texto[5..].

```
WriteLine("--- últimas 4 ---");
var Ultimas4 = texto[5..]; // 4 últimas pelo índice 5
foreach (var p in Ultimas4)
{
   WriteLine(p);
}

ReadLine();
```

Salve e execute F5 para ver o resultado. Caso queira colocar um *breakpoint* para analisar com calma, faça isto nos *loopings*.

***Resultado:***

—— Índices do dicionário ——

```
A última palavra é clientes
A primeira palavra é : Meu

—— Parte do texto ——

codigo
rodou
sem

—— Intervalo da coleção ——

codigo
rodou
sem
erros
para

—— Primeiras 4 ——

Meu
codigo
rodou
sem

—— Últimas 4 ——

para
ler
cadastro
clientes
```

## DELEGATE FUNC <>

Este tópico eu considero um dos melhores recursos da linguagem C#, o uso de *Delegates*. Basicamente temos situações onde é preciso disparar um código que será executado em tempo de execução, de acordo com a situação. Pense num conjunto de métodos assinados e prontos para uso, só que a chamada será gerada em tempo de execução apenas. Você não terá a notação verbosa, clara, detalhando todos os procedimentos passo a passo, as coisas estarão encapsuladas num conjunto de instruções a serem geradas dinamicamente!

Parece confuso, então vou tentar explicar com um exemplo simples e funcional. Todos os proprietários de carros, quando é preciso fazer a revisão no mecânico, é preciso seguir um checklist de vários itens. Quando o mecânico conecta o computador ao carro para diagnosticar problemas no motor, existem diversos blocos de códigos (podemos entender como métodos) que podem ou não ser disparados de acordo com o problema. Por exemplo, checar pressão dos pneus, caso estiver baixo, é chamado o método para calibrar o pneu, o qual recebe como parâmetro o tipo de carro, características do pneu, etc, para então saber quanto de pressão calibrar.

Já em casos de revisão do óleo do motor, é preciso disparar o código de verifica a viscosidade do óleo, a quantidade existente, a espessura do mesmo, etc e define se está na hora ou não de efetuar a troca. Caso esteja, é preciso disparar um outro código para checar o filtro do óleo, outro código para o filtro do ar condicionado. Enfim, note que dependendo do fluxo dos resultados, são disparados diversos blocos de códigos, rotinas, métodos, passando diversos parâmetros.

A estes blocos que existem e podem ou não ser disparados em tempo de execução é o que entendemos como *Delegates*. Eles estão sempre prontos para serem as vezes criados e executados em tempo de execução.

No C# temos o comando *FUNC* que é um *delegate* que tem até 16 entradas (parâmetros) e apenas uma saída (resultado).

Vamos a um exemplo prático, adicione o arquivo *ExemploFunc*.cs contendo a seguinte estrutura:

```
using System;
using System.Linq;
using static System.Console;

namespace livrocsharp
{
    class ExemploFunc
    {
        static void Main(string[] args)
        {

        }
    }
}
```

No *Main*, vamos iniciar com um código que declara uma variável contador do tipo *int* (inteiro) com valor inicial de zero. Em seguida, pense num código para incrementar o valor de 1 a cada chamada! Provavelmente você criaria um método para receber o valor atual, somar 1 e retornar o novo valor. Note como que o *delegate FUNC* é criado, temos a declaração *Func* seguida do tipo de dado de saída <*int*> (inteiro). Observe que não há nenhuma entrada de dados, só a saída, afinal o *Func* com apenas uma declaração significa o tipo de dado de saída, que é obrigatório.

Em seguida, temos a variável *numero*, que podemos entender como o nome do método a ser criado em tempo de execução. Em seguida, como não há parâmetros de entrada, não há o que informar entre os parênteses, ficando assim em branco, apenas o ( ). Agora vem o melhor de tudo, qual é a fórmula, o cálculo a ser pro-

cessado? Isto é definido após o lambda =>, neste caso ++contador, ou seja, a variável contador é ela mais um.

O resultado do primeiro *WriteLine* mostra o valor 1 na primeira chamada, o valor 2 na segunda chamada, afinal à variável contador foi adicionado o número 1. Ao final, é mostrado o conteúdo da variável contador, só para provar o conteúdo.

```
// Func< até 16 entradas e UMA saída >
int contador = 0;
Func<int> numero = () => ++contador;
WriteLine($"chamada 1 {numero()}");
WriteLine($"chamada 2 {numero()}");
WriteLine($"contador: {contador}");
```

Salve e execute para ver o resultado, em seguida coloque um *breakpoint* na linha que chama o método numero() e veja, que via *Reflection*, o método é mostrado na execução:

***Resultado:***

**chamada 1: 1**
**chamada 2: 2**
**contador: 2**

Neste próximo exemplo, antes de ver o código, como que você resolveria a seguinte questão: Quantos caracteres contém o texto "Visual C#"?

Imagino que você criaria um método que recebe como parâmetro o texto completo, conta a quantidade de caracteres e retorna o número. Este é o procedimento normal para um código explícito, verboso e de fácil entendimento.

No entanto, veja como que fica o mesmo código usando o *Func*. Sim, é apenas uma única linha contendo a declaração do *Func*<entrada, saída>, neste caso, a entrada é do tipo *string* e a saída, o re-

sultado é do tipo *int* (inteiro), contendo a quantidade de caracteres do texto. Em seguida temos o nome do método, neste caso, *MetodoQtdeCaracteres*, a declaração do parâmetro de entrada (t), de texto, seguido do cálculo a ser feito => *t.Lenght*. Ou seja, pegue o texto passado como parâmetro, aplique o método *Lenght* (tamanho) e retorne o número com a quantidade de caracteres.

Pronto, o *delegate* está criado e pronto para ser chamado no *WriteLine* na linha a seguir, contendo o texto a ser avaliado.

```
// Quantos caracteres contém o texto "Visual C#"?
Func<string, int> MetodoQtdeCaracteres = (t) => t.
Length;
WriteLine(MetodoQtdeCaracteres("Visual C#"));
```

Parece complicado, mas garanto que com o tempo isto se tornará mais fácil de entender, pois em apenas uma única linha temos tudo o que precisamos. E com o tempo, a manutenção e o entendimento fica melhor, deixando o código mais limpo, simples e de fácil manutenção.

Antes de executar o código, vou propor outro exercício: Quantas palavras contém uma expressão? E se tivermos muitos espaços em branco entre as palavras, como contar sem isto?

Usando o *delegate Func* teremos o seguinte código: *Func<string, int>* a entrada é *string* e a saída é um *int* (inteiro), o método chama *MetodoQtdePalavras*, o parâmetro é o t de texto. Já o código que efetiva a contagem de palavras, primeiro fazemos uma divisão de elementos usando o *Split*, pesquisando pelo caractere em branco (espaço). O *Split* cria uma coleção de itens num *array* com cada palavra.

Em seguida, o uso do *Where* com a condição != (diferente) de branco, ou seja, pega somente os textos. E, para finalizar usamos o *Count* para contar a quantidade de elementos que atendam esta condição.

```
// Qtas palavras contém uma expressão?
WriteLine("------- qtde palavras");
Func<string, int> MetodoQtdePalavras = (t) =>
        t.Split(' ')
        .Where(x => x != "")
        .Count();
WriteLine(MetodoQtdePalavras("  Brasil      campeã
o   mundial de   volei  "));
```

Salve e execute novamente e veja o resultado:

**Resultado:**

**—— Uso do Func: QtdeCaracteres ——**
**9**
**—— Uso do Func: QtdePalavras ——**
**5**

No próximo exemplo vamos aprimorar o uso do *Func* de maneira que tenhamos 3 entradas de dados e uma saída, todos do tipo *decimal*. O que faremos é um código para calcular o Imposto, tendo como entradas os valores dos salários, o percentual de desconto e a alíquota. Existe uma condição que se o salário for menor ou igual a 1000, o Imposto é zero, caso contrário, é calculado o percentual e subtraída a alíquota.

Vamos por parte, primeiro temos o *Func<decimal, decimal, decimal, decimal>* com as quatro variáveis do tipo *decimal*. Em seguida, o nome do método Imposto, neste caso. E o código atribuído => se o salario <= 1000, o retorno é 0; caso contrário a fórmula: salario * (perc / 100) – alíquota.

Já na chamada do *delegate Func* Imposto são passados os valores respectivamente.

```
// calcular o imposto
// salario, perc, aliquota
```

```
// formula: salario * (perc / 100) - aliquota
// condicao: salario <= 1000 >>> imposto = 0
// cond ? true : false
Func<decimal, decimal, decimal, decimal>
   Imposto = (salario, perc, aliquota) =>
   {
      return salario <= 1000 ? 0 :
      salario * (perc / 100) - aliquota;
   };
WriteLine("---- calculo do Imposto ---");
WriteLine(Imposto(1000, 10, 10));
WriteLine(Imposto(5000, 27.5M, 80));
WriteLine(Imposto(23500, 32.5M, 180));
```

Salve e execute para visualizar o resultado, e mais uma vez, vale inserir um *breakpoint* e avaliar a expressão passo a passo.

***Resultado:***

**—— Cálculo do Imposto ——**
**0**
**1295,000**
**7457,500**

## USO DO FUNC EM COLEÇÕES

Neste exemplo a seguir, iremos criar um *delegate Func* com um código que irá ler uma coleção de carros e aplicar o *delegate* com o respectivo código. Primeiro precisamos adicionar um novo arquivo chamado Carro.cs contendo o seguinte código. Nele, temos a classe Carro definida com todas as propriedades, algumas contendo fórmulas, por exemplo, *Consumo*(), *ConsumoString*() e *Idade*(), assim como o método *Get* que retorna uma lista de carros, usando *Generics*<*Carro*>*.* Note que o *Get* está como estático (*static*) para facilitar a chamada.

Cabe ressaltar que a propriedade Cor é um *enum* com as respectivas cores, assim facilita a entrada de dados e assegura que as cores sejam únicas.

```
using System;
using System.Collections.Generic;

namespace livrocsharp
{
   public class Carro
   {
      public int ID { get; set; }
      public string Modelo { get; set; }
      public int Litros { get; set; }
      public double KmRodados { get; set; }
      public int AnoFabricacao { get; set; }
      public string Cor { get; set; }

       public double Consumo() => KmRodados / Litro
s;
      public string ConsumoString() =>
            Litros > 0 ? $"{Consumo():n2}" : "----";
               public  int  Idade()  =>  DateTime.Today.
Year - AnoFabricacao;

       public override string ToString() => $"{ID} - {Mo
delo}";

      public static List<Carro> Get()
      {
         return new List<Carro>
         {
              new Carro { ID = 1, Modelo = "Honda civic"
, AnoFabricacao = 2000 , KmRodados = 22000, Cor = Co
res.Azul.ToString(), Litros=24 },
               new Carro { ID = 2, Modelo = "Ford GT", A
noFabricacao = 2015, KmRodados = 14000, Cor = Cores.
```

```
Verde.ToString(), Litros=32  },
    new Carro { ID = 3, Modelo = "BMW X1", AnoFabricacao = 2017, KmRodados = 1000, Cor = Cores.Vermelho.ToString(), Litros=23 },
    new Carro { ID = 4, Modelo = "Honda Fit", AnoFabricacao = 2000, KmRodados = 150600, Cor = Cores.Azul.ToString(), Litros=22 },
    new Carro { ID = 5, Modelo = "Ford Landau", AnoFabricacao = 1980, KmRodados = 150520, Cor = Cores.Azul.ToString(), Litros=20 },
    new Carro { ID = 6, Modelo = "BMW X5", AnoFabricacao = 2017, KmRodados = 3500, Cor = Cores.Vermelho.ToString(), Litros=15 },
    new Carro { ID = 7, Modelo = "Jaguar S1", AnoFabricacao = 2007, KmRodados = 6000, Cor = Cores.Azul.ToString(), Litros=10 },
    new Carro { ID = 8, Modelo = "Volks Polo", AnoFabricacao = 2014, KmRodados = 5090, Cor = Cores.Verde.ToString(), Litros=12 },
    new Carro { ID = 9, Modelo = "Ferrari F40", AnoFabricacao = 2010, KmRodados = 2000, Cor = Cores.Azul.ToString(), Litros=20 },
            };
        }
    }

    enum Cores
    {
        Vermelho = 1,
        Verde = 2,
        Azul = 3
    }
}
```

Salve este código da classe Carro.cs e retorne ao *ExemploFUNC.cs*. Adicione o código a seguir com objetivo de definir o *delegate Func CalculoKmPorAno*, carregar toda a lista de carros e disparar um

*looping* para mostrar o resultado de cada carro.

Existe uma condição no *Func* que se o ano do carro for maior ou igual ao ano atual do processamento, devemos mostrar apenas a km, caso contrário, aplicamos a fórmula de quilômetros médios rodados por ano, sendo km / (anoAtual – ano). Por exemplo, se o ano do carro for 2015 e o atual 2021, temos 6 anos. Se o carro rodou 6.000 / 6 anos, temos 1.000 km /ano. Perceba que isto é uma regra a ser aplicada, e deve ser para todos os carros, então, nada melhor que criar o *Func* e chamar quando preciso.

Como o ano atual não deve ser digitado, usamos o *DateTime.Today.Year* para capturar automaticamente o ano atual, neste caso, 2021.

No *Func* temos 2 entradas *int* (ano), *double* (km) e 1 saída *double*. O nome do método é *CalculoKmPorAno*, contendo os 2 parâmetros (ano, km) como entrada. A fórmula => verifica se o ano >= anoAtual, se sim, retorna apenas os km, caso contrário, calcula km / anos do carro.

Você deve estar se perguntando por que não uso o *DateTime.Today.Year* diretamente dentro do *Func*, no lugar da variável *anoAtual*? A resposta é: performance, na memória quando definimos uma variável, *anoAtual* neste caso, já é alocado um espaço com este conteúdo, ficando por referência. Como usamos este conteúdo mais de uma vez, multiplicado pela quantidade de carros na coleção, a performance aumenta muito. Imagine se a quantidade de elementos da coleção crescer muito, já temos isto definido por referência.

Pronto, o *Func* está definido, resta lermos todos os carros e aplicar o *looping*. Para isto, basta criar a variável *dados* do tipo *var (System.Collections.Generic.List<T>)* e atribuir o método *Get*() da classe Carro.

Em seguida, precisamos percorrer toda a lista de carros e exibir uma listagem com as propriedades (Modelo, Ano, Km, Cor, Litros,

km/ano). Claro que na classe Carro já temos os métodos Consumo e Idade que retorna praticamente a mesma coisa que o *Func*, mas a ideia aqui é criar um *Func*.

Veja o uso de *dados.ForEach*, já que dados é do tipo *Generics*, permite o uso do *ForEach* que irá disparar uma *Action*. A cada iteração do *looping* temos atribuído à variável kmPorAno, o Func CalculoKmPorAno, passando as propriedades AnoFabricacao e KmRodados como parâmetro.

Em seguida, no *WriteLine* são mostradas as demais propriedades do carro.

```
WriteLine("---- cálculo de FUNC em coleções");
// Fórmula: média de km rodados por ano
// km / (today.year - ano)
//"Modelo - Ano - Km - Cor - Litros - km/ano"
int anoAtual = DateTime.Today.Year;
Func<int, double, double> CalculoKmPorAno = (ano, km) =>
    ano >= anoAtual ? km : km / (anoAtual - ano);

// Carrega a lista de carros
var dados = Carro.Get();

// looping para varrer todos os carros da lista
dados.ForEach(x =>
{
    var kmPorAno = CalculoKmPorAno(
        x.AnoFabricacao, x.KmRodados);
            WriteLine($"{x.Modelo} | ano: {x.AnoFabricacao} | km: {x.KmRodados:n0} | litros {x.Litros} | km/ano: {kmPorAno:n0}");
});

ReadLine();
```

Salve e execute, veja a listagem de todos os carros.

*Resultado:*

—— cálculo de FUNC em coleções ——

Honda civic | ano: 2000 | km: 22.000 | litros 24 | km/ano: 1.048
Ford GT | ano: 2015 | km: 14.000 | litros 32 | km/ano: 2.333
BMW X1 | ano: 2017 | km: 1.000 | litros 23 | km/ano: 250
Honda Fit | ano: 2000 | km: 150.600 | litros 22 | km/ano: 7.171
Ford Landau | ano: 1980 | km: 150.520 | litros 20 | km/ano: 3.671
BMW X5 | ano: 2017 | km: 3.500 | litros 15 | km/ano: 875
Jaguar S1 | ano: 2007 | km: 6.000 | litros 10 | km/ano: 429
Volks Polo | ano: 2014 | km: 5.090 | litros 12 | km/ano: 727
Ferrari F40 | ano: 2010 | km: 2.000 | litros 20 | km/ano: 182

# CAPÍTULO 8 – INTRODUÇÃO AO LINQ

Neste capítulo, conheceremos um pouco sobre *LINQ (Language Integrated Query - Consulta Integrada à Linguagem)*, esse recurso foi inicialmente adicionado à versão do .NET Framework 3.5 no final do ano de 2007 juntamente com o Visual Studio 2008. Esse foi um marco muito importante para o ecossistema .NET, e desde então, tem ganhado melhorias significativas.

O LINQ surgiu para fornecer a capacidade de efetuar consultas em fonte de dados diferentes, seu principal objetivo é abstrair toda complexidade ao fazer consultas em uma fonte de dados, seja ela um XML, Banco de dados, ou até meio uma cadeia de caracteres. Antes do surgimento do LINQ, filtrar dados em uma fonte de dados como, por exemplo, um ARRAY, precisava de um certo conhecimento, assim era necessário criar um algoritmo que atendesse os critérios de seu filtro.

Consultar dados em diferentes fontes ficou muito mais fácil, dado que a complexidade abstraída pelo LINQ não nos obriga a criar consultas específicas para diferentes fontes de dados. Por exemplo, se você está escrevendo uma aplicação que se conecta a um banco de dados relacional, não necessariamente você precisa aprender *SQL (Linguagem de Consulta Estruturada)* para fazer suas consultas, da mesma forma para consultar dados em um XML, você não precisar ficar percorrendo todos os nós do documento, o LINQ já fornece a melhor experiência para fazer consultas, seja ela em objetos, XML ou para qualquer coleção de objeto que forneça à você suporte ao *IEnumerable* ou a interface genérica *IEnumerable*<*T*>.

O LINQ aplica filtros em consultas por meio de expressões ou delegados (*delegate*), em tempo de desenvolvimento podemos explicitamente especificar quais são os filtros e critérios desejados para serem aplicados em sua consulta. Quando fazemos uma consulta sobre um *IEnumerable<T>*, significa que a consulta será compilada para um *delegate*, mas se fizermos uma consulta sobre um *IQueryable<T>*, ela será compilada para uma árvore de expressões. Isto possibilita que algum manipulador possa extrair informações de sua expressão e consiga tomar alguma decisão sobre o que fazer com o filtro utilizado em sua consulta. Um bom exemplo são os *ORM's (Modelos de Objetos Relacionais)*, como o popular *Entity Framework Core*.

Podemos escrever suas consultas LINQ de duas formas: a mais comum é a "sintaxe de consulta", dado que é mais expressiva, muito mais próximo do que um DBA, por exemplo, poderia escrever para executar uma consulta em um banco de dados, "sintaxe de consulta" foi basicamente inspirada em uma linguagem natural, onde o próprio texto se explica e de fácil entendimento; podemos também fazer consultas por meio da "sintaxe de método", como o próprio nome já diz, basicamente utiliza métodos para fazer consultas.

E qual a diferença? Em termos de performance não existe nenhuma diferença, dado que quando seu programa é compilado, ambas produzem o mesmo resultado, mas, existe uma bifurcação aqui para tomadas de decisões. Ao utilizar "sintaxe de consulta" realmente torna o código mais expressivo, mas existem algumas limitações  ao utilizar esse tipo de consulta, por exemplo, algumas operações não conseguiremos fazer como Min, Max, Count, Sum, forçando mesclar a consulta utilizando a "sintaxe de método". Sendo assim, é necessário conhecer melhor ambos tipos de consultas suportado pelo LINQ, mas já posso lhe garantir que usando "sintaxe de método" teremos muito mais flexibilidade.

O LINQ realmente nos proporciona uma excelente experiência ao consultar informações em uma fonte de dados, além de efetuar consultas, conseguimos manipular o resultado, ou seja, transfor-

mar os dados do resultado da consulta para entregar os dados formatados ao consumidor. Para ficar mais didático, pense no cenário onde fazemos uma consulta em um banco de dados e transformamos o resultado materializado de uma consulta em um XML, ou até mesmo uma string formatada para ser gravada em um arquivo CSV.

Antes de mergulhar profundamente no LINQ é preciso conhecer a diferença entre IEnumerable<T> e IQueryable<T>, isso vai ser importante para entender melhor o funcionamento dos filtros que serão aplicados nas consultas pelo LINQ.

IEnumerable<T> tem como objetivo expor um enumerador para que possamos iterar sobre a instância de um objeto. Cabe ressaltar que todo processamento é executado na memória, os métodos que possuem sobrecargas aguardam um delegado do tipo Func<T, TResult> onde sempre é fornecido um tipo de entrada e um tipo de saída, conforme a figura a seguir.

```
var numeros = new[] { 1, 2, 3 };
// Função (entrada inteiro, saída boleano)
Func<int, bool> funcao = i => i > 2;
var resultado = numeros.Where(funcao);
// Faz iteração na variável resultado
foreach (var numero in resultado)
{
    // Imprime número
    Console.WriteLine(numero);
}
// Resultado:
3
```

Figura 24 - Estrutura do IEnumerable

Vamos entender o que o código acima está fazendo. Primeiramente criamos uma coleção de números de 1 a 3, em seguida criamos uma função que valida se um número é maior que dois, por meio de uma expressão *lambda*, e assim aplicamos um filtro usando o método ***where*** o qual iremos abordar sobre ele logo mais.

Depois fizemos uma iteração sobre o enumerador da variável resultado e imprimimos o resultado no console da aplicação, veja uma ilustração na figura a seguir.

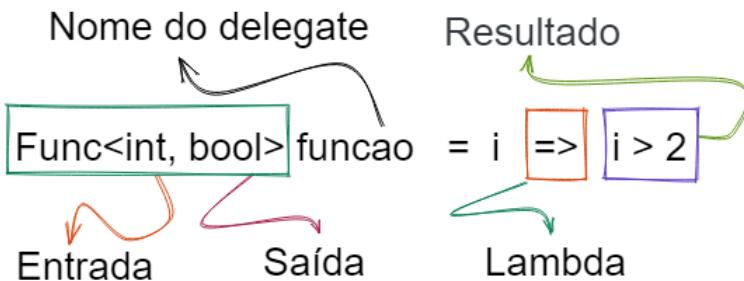



Figura 25 - Iteração do código

A figura anterior ilustra melhor a anatomia de um filtro em uma consulta com LINQ. Informamos o valor de entrada que queremos interagir, em nosso exemplo da coleção de inteiros, será informado um valor do tipo *int*, e o resultado é um *booleano*, dado que queremos validar se o valor imputado é maior que dois, sendo assim quando o LINQ executar seu algoritmo para filtrar os elementos da coleção e atender o filtro de saída, ele irá selecionar o elemento como resultado.

*IQueryable<T>* tem como objetivo disponibilizar os recursos e todo poder do LINQ para que os escritores de provedores de acesso a dados possam ser capazes de escrever instruções SQL para serem executadas em banco de dados relacional, com base nas expressões fornecidas pelo desenvolvedor.

Não iremos nos aprofundar no *IQueryable<T>*, dado que nosso objetivo é apenas apresentar conceitos do LINQ, mas vamos entender um pouco sobre seu comportamento. O *IQueryable<T>* diferentemente do *IEnumerable<T>* espera em seus métodos expressões do tipo *Expression<Func<T,TResult>>*, é dessa forma que os escritores de provedores que desejam fazer implementações de consultas são capaz de extrair da árvore de expressão os parâmetros sufi-

cientes para montar uma instrução SQL que possa ser executada em um banco de dados relacional, por exemplo, um provedor que faz bastante uso do *IQueryable<T>* é o famoso *ORM Entity Framework Core*.

```
var numeros = new[] { 1, 2, 3 }.AsQueryable();
// Expressão
Expression<Func<int,bool>> funcao = p => p > 2;
// Filtro
var resultado = numeros.Where(funcao);
// Faz iteração na variável resultado
foreach (var numero in resultado)
{
    // Imprime número
    Console.WriteLine(numero);
}
// Resultado:
//3
```

Figura 26 - Expressão

Ao observarmos a figura anterior, a única diferença encontrada com uma consulta de uma coleção que herda o tipo *IEnumerable<T>*, é que ao invés de passar apenas um delegado (*delegate*), agora estamos passando uma expressão com um delegado (*delegate*). É exatamente por meio dessas expressões que os criadores de ORM's ou escritores de provedores conseguem identificar pela árvore de expressão que tipo de dados estamos consultando, quais operadores utilizamos e que tipo de retorno esperamos. A seguir iremos falar um pouco mais sobre consultas LINQ, aquela que é processada apenas na memória, como já explicado anteriormente.

Quando escrevemos uma consulta LINQ ela é executada automaticamente em tempo de execução? A resposta é não, ela será processada somente quando fizermos uma iteração por meio do *foreach* ou usar algum método de extensão como por exemplo o *ToList* ou *ToArray*.

## CRIANDO A PRIMEIRA CONSULTA COM LINQ

Vamos colocar em prática toda teoria adquirida até aqui, sendo assim usaremos como um exemplo hipotético um array com dez números, onde precisamos filtrar os números que são maiores que cinco e imprimir o resultado na tela, conforme figura a seguir.

```
1  // Array de números
2  var numeros = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
3
4  // Filtrar números
5  var numerosFiltrados = numeros.Where(n=> n > 5);
6
7  // Fazer iteração
8  foreach(var numero in numerosFiltrados)
9  {
10     Console.WriteLine(numero);
11 }
12
13 // Resultado:
14 6
15 7
16 8
17 9
18 10
```

Figura 27 - Sintaxe de método

Observamos como foi simples fazer uma consulta usando **LINQ** em uma fonte de dados. Novamente, para reforçar os conceitos, a consulta só foi processada exatamente no momento que foi feito a iteração sobre a variável ***numerosFiltrados*** por meio do **foreach,** e não ao usar o método de extensão Where. *T*ambém foi usado o estilo de consulta **“sintaxe de método”**, mas podemos usar **“sintaxe de consulta”** da qual já falamos no início deste capítulo, conforme figura a seguir, o resultado é o mesmo.

```
1  // Array de números
2  var numeros = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
3
4  // Filtrar números (sintaxe consulta)
5  var numerosFiltrados =
6          from n in numeros where n > 5 select n;
7
8  // Fazer iteração
9  foreach(var numero in numerosFiltrados)
10 {
11     Console.WriteLine(numero);
12 }
13
14 // Resultado:
15 6
16 7
17 8
18 9
19 10
```

Figura 28 - Sintaxe de consulta

## OPERADORES SUPORTADOS PELO LINQ

O LINQ possui inúmeros operadores de consultas, alguns são suportados apenas por meio de “sintaxe de método”, ou seja, métodos de extensões. Os operadores mais comuns são: projeção de dados, restrições, junções, ordenações, agrupamentos, agregações e paginações.

| Métodos de projeção e restrição | | |
|---|---|---|
| Select | SelectMany | Where |

| Métodos de ordenação | | |
|---|---|---|
| OrderBy | OrderByDescending | ThenBy |
| ThenByDescending | Reverse | |

| Métodos de Agregação | | |
|---|---|---|
| Average | Count | LongCount |
| Max | Min | Sum |

| Métodos de paginação | | |
|---|---|---|
| First | FirstOrDefault | Last |
| LastOrDefault | Single | SingleOrDefault |
| Skip | SkipWhile | Take |
| TakeWhile | | |

| Métodos de junção | |
|---|---|
| Join | GroupJoin |

## CONHECENDO OS OPERADORES DO LINQ

Neste tópico, iremos conhecer melhor os operadores suportados pelo LINQ, a ordem que será apresentada é do menos ao mais complexo. O primeiro será o Where que um operador de restrição, podendo ser usado tanto na “sintaxe de consulta” quanto na “sintaxe de método”. Nos exemplos, usaremos a “sintaxe de método” dado que fornece um melhor suporte.

## OPERADOR WHERE

Recebe os critérios que devem ser aplicados em sua consulta, interpreta e filtra os elementos com base nos critérios informados pelo desenvolvedor. Veja um exemplo onde temos uma coleção de nomes de pessoas e precisamos filtrar pessoas que contém um determinado nome, que é Almeida, conforme figura a seguir.

```
// Coleção de pessoas
var pessoas = new[]
{
    "Carlos dos Santos",
    "Renato Haddad",
    "Claudenir Andrade",
    "Andre Carlucci",
    "Ray Carneiro",
    "Rafael Almeida",
};

// Filtro de pessoas que contém nome Silva
var pessoasFiltradas = pessoas.Where(p=> p.Contains("Almeida"));

// Iteração com as pessoas filtradas
foreach(var pessoa in pessoasFiltradas)
{
    // Imprime no console o nome da pessoa
    Console.WriteLine(pessoa);
}

// Resultado:
Rafael Almeida
```

Figura 29 - Código com o operador Where

## OPERADOR DE AGREGAÇÃO – COUNT

Fornece a capacidade de efetuar uma contagem de elementos em uma coleção, podemos contar todos elementos ou informar um critério que desejamos aplicar para o LINQ filtrar uma determinada região dos dados em sua coleção, conforme figura a seguir.

```
1  var numeros = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
2
3  // Faz contagem em toda coleção
4  var countTotal = numeros.Count();
5
6  // Faz contagem na coleção com filtro
7  var countParcial = numeros.Count(n => n > 5);
8
9  // Imprime Contagem
10 Console.WriteLine(countTotal);
11 Console.WriteLine(countParcial);
12
13 // Resultado:
14 10
15 5
```

Figura 30 - Código com o operador Count

## OPERADOR DE AGREGAÇÃO – SUM

Fornece a capacidade de efetuar cálculo de soma sobre os elementos da coleção, é limitado a tipos **int, double, decimal, float e long,** conforme figura a seguir.

```
1  var numeros = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
2
3  // Soma todos elementos da coleção
4  var somaTotal = numeros.Sum();
5
6  // Soma o elemeto da coleção multiplicado por 2
7  var somaComMultiplicacao = numeros.Sum(n => n * 2);
8
9  // Imprime Soma
10 Console.WriteLine(somaTotal);
11 Console.WriteLine(somaComMultiplicacao);
12
13 // Resultado:
14 55
15 110
```

Figura 31 - Código com o operador Sum

## OPERADOR DE AGREGAÇÃO – MAX

Recupera o valor máximo de um elemento da coleção. Para os métodos de agregação não existe a necessidade de fazer iteração, dado que seu valor é retornado imediatamente pelo **LINQ**, após sua chamada**,** conforme figura a seguir.

```
1  var numeros = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
2
3  // Recupera o valor máximo de um elemento da coleção
4  var maxTotal = numeros.Max();
5
6  // Valor máximo do elemeto da coleção multiplicado por 2
7  var maxComMultiplicacao = numeros.Max(n => n * 2);
8
9  // Imprime valor máximo
10 Console.WriteLine(maxTotal);
11 Console.WriteLine(maxComMultiplicacao);
12
13 // Resultado:
14 10
15 20
```

Figura 32 - Código com o operador Max

## OPERADOR DE AGREGAÇÃO – MIN

Recupera o valor mínimo de um elemento da coleção, assim como o operador *Max*, o *Min* também possui método de sobrecarga, conforme figura a seguir.

```
var numeros = new[] { 1, 2, 3, 4 };

// Executa o operador de paginação take na coleção
var doisElementos = numeros.Take(2);

// Faz iteração sobre a variável doisElementos
foreach (var numero in doisElementos)
{
    // Imprime número
    Console.WriteLine(numero);
}
// Resultado:
1
2
```

Figura 33 - Código com o operador Min

## OPERADOR DE AGREGAÇÃO – AVERAGE

É responsável por efetuar o cálculo da média aritmética de uma sequência numérica dos itens de uma coleção. É importante ressaltar que para elementos da coleção do tipo **int** e **long** o valor devolvido pelo método será do tipo **double**, mas se os itens da coleção forem do tipo **decimal**, **float** ou **double** o próprio método de extensão já devolve o valor com base em seu respectivo tipo de dados, conforme figura a seguir.

```
var numeros = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Média dos elementos da coleção
var media = numeros.Average();

// Imprime valor da média
Console.WriteLine(media);

// Resultado:
5.5
```

Figura 34 - Código com o operador Average

## OPERADOR DE ORDENAÇÃO – REVERSE

Este operador fornece a capacidade de devolver uma coleção em uma ordem inversa, conforme figura a seguir.

```
var numeros = new[] { 3, 1, 2 };
// Executa o operador de ordenação na coleção
var numerosOrdenados = numeros.OrderByDescending(n => n);
// Faz iteração sobre a variável numerosOrdenados
foreach (var numero in numerosOrdenados)
{
    // Imprime número
    Console.WriteLine(numero);
}
// Resultado:
3
2
1
```

Figura 35 – Código com o operador Reverse

## OPERADOR DE ORDENAÇÃO – ORDERBY

Faz a ordenação dos elementos em ordem ascendente (do menor para o maior) respeitando o tipo de objeto da coleção, conforme figura a seguir.

```
 1 var numeros = new[] { 3, 1, 2 };
 2 // Executa o operador de ordenação na coleção
 3 var numerosOrdenados = numeros.OrderBy(n => n);
 4 // Faz iteração sobre a variável numerosOrdenados
 5 foreach (var numero in numerosOrdenados)
 6 {
 7     // Imprime número
 8     Console.WriteLine(numero);
 9 }
10
11 // Resultado:
12 1
13 2
14 3
```

Figura 36 - Código com o operador ordem ascendente

## OPERADOR DE ORDENAÇÃO – ORDERBYDESCENDING

Faz a ordenação dos elementos em ordem descendente (do maior para o menor) respeitando o tipo de objeto da coleção, conforme figura a seguir.

```
 1 var numeros = new[] { 3, 1, 2 };
 2 // Executa o operador de ordenação na coleção
 3 var numerosOrdenados = numeros.OrderBy(n => n);
 4 // Faz iteração sobre a variável numerosOrdenados
 5 foreach (var numero in numerosOrdenados)
 6 {
 7     // Imprime número
 8     Console.WriteLine(numero);
 9 }
10
11 // Resultado:
12 1
13 2
14 3
```

Figura 37 - Código com o operador ordem descendente

## OPERADOR DE PAGINAÇÃO – TAKE

Este operador de paginação, também conhecido como operador de particionamento, devolve uma nova coleção de objetos com a quantidade de elementos informados, respeitando todos critérios de filtros aplicados a sua consulta, conforme figura a seguir.

```
var numeros = new[] { 5, 4, 9, 6 , 7, 2};
// Executa o operador de paginação TakeWhile na coleção
var items = numeros.TakeWhile(n => n < 6);
// Faz iteração sobre a variável items
foreach (var numero in items)
{
    // Imprime número
    Console.WriteLine(numero);
}
// Resultado:
5
4
```

Figura 38 - Código com o operador Take

## OPERADOR DE PAGINAÇÃO – TAKEWHILE

O operador de paginação TakeWhile funciona como um filtro, ele faz iteração na coleção enquanto atender os critérios fornecidos, ou seja, quando a condição for falsa ele devolve os elementos da coleção que atenderam os critérios. Com isso é interrompido o ciclo de iteração com a coleção. Neste exemplo temos uma coleção de números aleatórios e queremos que retorne os elementos até ele detectar que o valor do elemento é inferior a seis. Mas isso não é o mesmo comportamento do *Where*? Não, porque o *Where* continua a iterar e filtrar os dados até o último elemento da coleção, conforme figura a seguir.

```
var numeros = new[] { 1, 2, 3, 4 };

// Executa o operador de paginação take na coleção
var doisElementos = numeros.Take(2);

// Faz iteração sobre a variável doisElementos
foreach (var numero in doisElementos)
{
    // Imprime número
    Console.WriteLine(numero);
}
// Resultado:
1
2
```

Figura 39 - Código com o operador TakeWhile

## OPERADOR DE PAGINAÇÃO – FIRST

O método de *First*() retorna o primeiro elemento da coleção, conforme figura a seguir.

```
var numeros = new[] { 5, 4, 9, 6 , 7, 2};
// Executa o operador de paginação First na coleção
var numero = numeros.First();
// Imprime número
Console.WriteLine(numero);
// Resultado:
//5
```

Figura 40 - Código com o operador First

O *First* possui também uma sobrecarga possibilitando aplicar um filtro em nossas consultas, assim que o primeiro elemento da lista atender os critérios será imediatamente selecionado e devolvido, conforme figura a seguir.

```
var numeros = new[] { 5, 4, 9, 6 , 7, 2};
// Executa o operador de paginação First na coleção
var numero = numeros.First(p => p == 9);
// Imprime número
Console.WriteLine(numero);
// Resultado:
9
```

Figura 41 - Código com o operador First

Para ambos métodos com e sem sobrecarga será gerado uma exceção do tipo **InvalidOperationException**, informando que não existe item na coleção ou não encontrou um elemento que atendesse os critérios informados, conforme figura a seguir.

```
Exception has occurred: CLR/System.InvalidOperationException ×
An unhandled exception of type 'System.InvalidOperationException'
occurred in System.Linq.dll: 'Sequence contains no matching element'
   at System.Linq.ThrowHelper.ThrowNoMatchException()
   at System.Linq.Enumerable.First[TSource](IEnumerable`1 source,
Func`2 predicate)
```

Figura 42 - Exceção

## OPERADOR DE PAGINAÇÃO – FIRSTORDEFAULT

O método *FirstOrDefault* basicamente tem o mesmo comportamento do método *First*, a única diferença é que não será gerada uma exceção em caso da coleção estar vazia ou não ser encontrado o elemento que atenda ao critério passado na sobrecarga do método. Neste caso, o *FirstOrDefault* devolve o valor padrão, um exemplo, se for uma coleção de inteiros será devolvido o valor **0**, se for de objetos será devolvido *null*, conforme figura a seguir.

```
var numeros = new[] { 5, 4, 9, 6 , 7, 2};
// Executa o operador de paginação FirstOrDefault na coleção
var numero = numeros.FirstOrDefault(p=>p == 10);
// Imprime número
Console.WriteLine(numero);
// Resultado:
0
```

Figura 43 - Código com o operador FirstOrDefault

A figura anterior usa uma expressão lambda para filtrar um elemento na coleção que seja igual a **10**, mas não existe este elemento em nossa coleção, dado esse cenário o método retorna o valor padrão correspondente ao tipo da coleção, nesse caso 0, porque a coleção é de inteiros.

## OPERADOR DE PAGINAÇÃO - LAST E LASTORDEFAULT

O LINQ também fornece métodos para ler o último elemento da coleção. Os métodos *Last* e *LastOrDefault* possuem sobrecarga possibilitando passar um predicado para filtrar o elemento da coleção. O comportamento é o mesmo do *First* e *FirstOrDefault*, o que muda é apenas o retorno do elemento, retornando o último elemento da coleção, veja figura a seguir.

```
var numeros = new[] { 5, 4, 9, 6 , 7, 2};
// Executa o operador de paginação LastOrDefault na coleção
var numeroDefault = numeros.LastOrDefault(p=>p == 10);
// Imprime número
Console.WriteLine(numeroDefault);
// Resultado:
0
// Executa o operador de paginação Last na coleção
var ultimo = numeros.Last();
// Imprime número
Console.WriteLine(ultimo);
// Resultado:
2
```

Figura 44 - Código com o operador Last

## OPERADOR DE PAGINAÇÃO - SKIP

O *Skip* é um recurso muito importante no LINQ fornecendo a capacidade de ler elementos a partir de uma determinada posição dentro da coleção. Geralmente é utilizado em paginação de dados, basicamente o que ele faz é pular elementos da coleção até chegar na posição desejada. Veja o exemplo, conforme figura a seguir.

```
var numeros = new[] { 1, 2, 3, 3 , 3, 4};
// Executa o operador de paginação Skip na coleção
var numerosPaginados = numeros.Skip(3);
// Faz iteração em numerosPaginados
foreach (var numero in numerosPaginados)
{
    // Imprime número
    Console.WriteLine(numero);
}

// Resultado:
3
3
4
```

Figura 45 - Código com o operador Skip

Neste código temos uma coleção de números (1, 2, 3, 3, 3 e 4), pedimos para pular 3 elementos da coleção, e foi exatamente o que aconteceu, os elementos *1, 2* e o primeiro 3 da coleção foram descartados, processando apenas o segundo número 3 em diante como podemos ver no resultado (*3, 3 e 4*).

## OPERADOR DE JUNÇÃO – CONCAT

Durante o processo de desenvolvimento de software, às vezes nos deparamos com um cenário onde é necessário concatenar 2 coleções. O **LINQ** fornece a capacidade de fazer isso de forma simples e prática, fornecendo ao desenvolvedor um desenvolvimento muito mais rápido onde lhe proporciona uma produtividade melhor. Veja um exemplo no qual temos duas coleções de números,

a primeira com os números (*1,2,3*) e a segunda com os números (*5,6,7*), conforme a seguir.

```
var colecao1 = new[] { 1, 2, 3};
var colecao2 = new[] { 5, 6, 7};
// Executa o operador de junção concat na coleção
var numerosConcatenados = colecao1.Concat(colecao2);
// Faz iteração em numerosConcatenados
foreach (var numero in numerosConcatenados)
{
    // Imprime número
    Console.WriteLine(numero);
}
// Resultado:
1
2
4
5
6
7
```

Figura 46 - Código com o operador Concat

## OPERADOR DE PAGINAÇÃO - ELEMENTAT

Para cenários onde precisamos recuperar um elemento pelo de índice, o LINQ fornece o método *ElementAt*, tendo os mesmos comportamentos do *First* e do *FirstOrDefault*, veja um exemplo na figura a seguir.

```
var numeros = new[] { 1, 2, 3};
// Executa o operador de paginação ElementAt na coleção
var numero = numeros.ElementAt(1);
// Imprime número
Console.WriteLine(numero);
// Resultado:
2
```

Figura 47 - Código com o operador ElementAt

*Observação*: O primeiro elemento de uma coleção ou array sem-

pre será na posição 0, dado isso o resultado é 2, porque está na posição 1 da coleção.

## OPERADOR DE PAGINAÇÃO – SINGLE E SINGLEORDEFAULT

*Single* e *SingleOrDefault* são recursos de paginação extremamente importantes, sendo úteis em alguns cenários. Os comportamentos são parecidos com o *First* e *FirstOrDefault*, que tem como objetivo retornar um único elemento de uma coleção, mas o *Single* e *SingleOrDefault* é mais robusto, ele consegue validar se existe na coleção mais de 1 (um) item que atende os mesmos critérios do filtro, se existir ele vai lançar uma exceção, informando que a coleção contém mais de um elemento, conforme figura a seguir.

```
var numeros = new[] { 1, 2, 2, 3 };
// Executa o operador de paginação Single na coleção
var numero = numeros.Single(p => p == 2);
```

Figura 48 - Código com o operador Single

Neste exemplo estamos tentando filtrar um elemento na coleção que corresponda ao número dois, mas como vemos ele encontrou dois elementos que correspondem ao filtro informado no método de sobrecarga, que nesse caso será lançada a exceção, conforme a seguir.

```
Exception has occurred: CLR/System.InvalidOperationException ×
An unhandled exception of type 'System.InvalidOperationException' occurred in
System.Linq.dll: 'Sequence contains more than one matching element'
```

Figura 49 - Exceção InvalidOperationException

Vamos agora para um cenário feliz, usando a mesma coleção anterior, aplicamos um filtro diferente.

```
var numeros = new[] { 1, 2, 2, 3 };
// Executa o operador de paginação Single na coleção
var numero = numeros.Single(p => p == 3);
// Imprime número
Console.WriteLine(numero);
// Resultado:
3
```

Figura 50 - Código com o operador Single Filtro com Sucesso

Este código demonstra como resultado o valor 3, isso significa que tivemos sucesso com o filtro aplicado e que o *Single* fez sua validação e encontrou apenas um elemento que corresponde ao filtro aplicado.

O uso desse recurso é bem comum quando é preciso  ter a garantia de unicidade, fazendo com que seja aplicada restrições de validação em seu filtro. Um exemplo constante de uso deste recurso são os ORM's, onde traduzem para o servidor que a consulta pode retornar 2 (registos), e do lado cliente, ou seja do lado da aplicação, o *Single* aplica suas validações.

## OPERADOR DE PROJEÇÃO – SELECT

O *Select* é um operador de projeção poderoso do LINQ, fornecendo a capacidade de transformar os objetos de sua coleção, em uma nova coleção de objetos formatados, enriquecidos ou mais filtrados, ou seja, você pode criar novos tipos de objetos, fornecendo infinitas possibilidades de manipulação de seus dados, veja um exemplo na figura a seguir.

```
var numeros = new[] { 1, 2, 3 };

// Executa o operador de projeção select na coleção
var categorias = numeros.Select(p =>
     new
     {
        Id = p, // p = número da coleção
        Descricao = $"Categoria {p}"
     });

// Faz iteração na variável categorias
foreach (var c in categorias)
{
     // Imprime categoria ;
     var descricao = $"Categoria: {c.Id}-{c.Descricao}";
     Console.WriteLine(descricao);
}

// Resultado:
Categoria: 1-Categoria 1
Categoria: 2-Categoria 2
Categoria: 3-Categoria 3
```

Figura 51 - Código com o operador Select

Neste exemplo temos uma coleção de números que foi utilizada para criar uma nova coleção, e com um novo tipo de objeto anônimo. O que fizemos foi basicamente usar o operador de projeção *select* e por meio de uma *lambda* criar um novo tipo de objeto. No exemplo, temos a variável *categorias* que recebe a nova coleção, em seguida fazemos uma iteração na variável imprimindo no console da aplicação o resultado da transformação feita pelo operador de projeção *select*.

# CAPÍTULO 9 – REMOVER USING, INTERPOLAÇÃO DE STRINGS E PROPAGAÇÃO DE NULOS EM COLEÇÕES

## COMO REMOVER O USING DE CLASSES ESTÁTICAS?

O .NET é composto de *namespaces* e classes, o qual temos acesso aos métodos e propriedades. Todos os arquivos .cs tem nas primeiras linhas as declarações dos *namespaces*.

No entanto, para deixar o código mais simples e de fácil entendimento, podemos declarar os *namespaces* como estáticos (*static*), o que facilita o uso dos métodos nos códigos.

Por exemplo, o uso do *Console*.*WriteLine* pertence ao *namespace System*. Imagine um código onde temos que mostrar muitas informações, ou seja, com vários usos do *Console*.*WriteLine*. E, que tal usarmos apenas *WriteLine*? Para isto, crie uma pasta chamada Cap9 e o arquivo *usingStatic*.*cs* dentro desta pasta.

Nas primeiras linhas declare os três *namespaces* como estáticos (*using static*). Desta forma, tudo que estiver dentro destes *namespaces*, podemos acessar apenas declarando o nome do método.

```
System.Math e System.Console
```

No SVM (*static void Main*) vamos mostrar o resultado (*WriteLine*) da raiz quadrada de um número, neste caso, o método é *Sqrt* e o

número é o resultado de 3*3 + 4*4. Aliás, o uso do *WriteLine* será usado sempre que precisarmos mostrar uma informação ao usuário.

Em seguida, precisamos exibir o valor absoluto de um número, o qual representa apenas o número, independente do sinal positivo ou negativo. Neste caso, o método é o *Abs*, seguido do respectivo número (positivo e negativo).

Já o método *Max* retorna o maior valor entre dois números, portanto, digite o código *Max*(100, 590). O mesmo vale para o método *Min*, que retorna o menos valor entre dois números, neste caso, *Min*(100, 590).

Em casos de números que necessitam ser arredondados, usamos o método *Round*(número, casas decimais). Veja o código onde temos os números decimais 3250.895 e 3250.899 para serem mostrados com arredondamento de 1 e 2 casas decimais.

```
using static System.Console;
using static System.Math;
using static System.String;

namespace livrocsharp
{
  class usingStatic
  {
    static void Main()
    {
      // System.Math
      // raiz quadrada
      WriteLine($"Raiz quadrada: {Sqrt(3*3 + 4*4)}");

      //  Retorna o valor absoluto
       WriteLine($"Valor Absoluto positivo: {Abs(1234
.78)}");
          WriteLine($"Valor Absoluto negativo: {Abs(-
```

```
1234.78)}");
            WriteLine($"Valor Absoluto negativo: {Abs(-850.99M)}");

         WriteLine($"Valor Máximo: {Max(100, 590)}");
         WriteLine($"Valor Mínimo: {Min(100, 590)}");

         // Arredonda um valor para o inteiro mais
   próximo ou para o número especificado de casas decimais.
          WriteLine($"Arredonda com 1 casa decimal: {Round(3250.895M, 1)}");
         WriteLine($"Arredonda com 2 casas decimais: {Round(3250.899M, 2)}");
       }
     }
   }
```

Salve o arquivo e antes de executa-lo, abra o arquivo *livrocsharp.csproj* e defina no *StartupObject* o nome do arquivo (classe) a ser aberto ao executar o projeto com F5.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
              <StartupObject>livrocsharp.usingStatic</StartupObject>
  </PropertyGroup>
</Project>
```

Agora sim, pressione F5 para ver o resultado do código.

***Resultado:***

**Raiz quadrada: 5**
**Valor Absoluto positivo: 1234,78**

```
Valor Absoluto negativo: 1234,78
Valor Absoluto negativo: 850,99
Valor Máximo: 590
Valor Mínimo: 100
Arredonda com 1 casa decimal: 3250,9
Arredonda com 2 casas decimais: 3250,90
```

### SYSTEM.STRING

Vamos aos exemplos dos métodos do *namespace System.String*. Para obter o tamanho de uma *string*, use o método *Length*, neste caso, retorna a quantidade de caracteres de uma expressão. Já o método *Concat* serve para juntar, concatenar duas strings.

Caso queira inserir uma *string* em uma expressão, use o *Insert*(posição, texto), sendo que a posição é o local onde iniciará, e texto é a expressão a ser adicionada. Neste caso, iremos adicionar o texto “avançado” na expressão contida na variável livro, a partir da posição 6. Ao final, o método *ToUpper* converte tudo para maiúsculo.

```
static void Main()
{
  ...

  // System.String
    WriteLine($"Tamanho da string: {"Livro de C#".Length}");
    WriteLine($"Concatena 2 strings: {Concat("Livro de C#, ", "Livro de EF Core")}");
  string livro = "Livro de EF Core";
   WriteLine($"{livro.Insert(6,"avançado ").ToUpper()}");
}
```

Salve e execute. Teremos o resultado a seguir.

***Resultado:***

```
Concatena 2 strings: Livro de C#, Livro de EF Core
LIVRO AVANÇADO DE EF CORE
```

Se olharmos o código no geral, está mais limpo, com menos informações e funcional. Isto se deve ao uso da declaração do *using static*.

## USO DE NAMEOF NO OPERADOR

Em certos casos onde temos diversas chamadas de métodos passando parâmetros, muitas vezes podem ocorrer erros, valores inválidos ou nulos, fórmulas erradas, etc. E, na maioria das vezes a mensagem de erro que retorna é genérica. Veja no capítulo sobre tratamento de erro como customizar a mensagem de erro específica.

Para estes casos de não deixar o usuário ou a área de suporte na mão, sem saber ao certo qual o parâmetro que está com erro, foi criado o *nameof*.

Na pasta do Cap 9, adicione um novo arquivo chamado *usoDoNameof.cs*, conforme o código a seguir. Primeiro temos um método chamado ***idade*** que recebe o ano (*pAno*) do tipo *int*, o qual verifica se o ano passado como parâmetro é maior > que o ano da data atual (*DateTime.Today.Year*). Caso verdadeiro, retorna uma exceção com a mensagem "ano inválido" contendo o nome do parâmetro *pAno*. Caso contrário, subtrai os anos e retorna o inteiro.

Note neste método que o parâmetro *pAno* está explícito dentro da expressão *nameof(pAno)*. Isto significa que em caso de erro, você saberá exatamente quem deu erro. Claro que aqui temos apenas um parâmetro, mas imagine em métodos com mais de 3 parâmetros, o desenvolvedor terá que adivinhar o erro.

Já no método SVM (*static void Main*) é feito o *try..catch* para tratar o erro. Aqui temos a *string* produto e a data de lançamento (lancto) preenchidas. No *WriteLine*, o método idade é chamado passado

apenas o ano do lançamento. Aqui não dará erro, mas no segundo *WriteLine* sim, pois o ano atual é 2001 e o passado é 2023.

```
using System;
using static System.Console;

namespace livrocsharp
{
  class usoDoNameof
  {
    static void Main()
    {
      try
      {
        string produto = "peixe";
         DateTime lancto = new DateTime( 2010, 5, 1
5 );
          WriteLine($"o {produto} tem {idade(lancto.
Year)} anos");
         WriteLine($"o {produto} tem {idade(2023)} a
nos");
      }
      catch (Exception ex)
      {
         WriteLine(ex.Message);
      }
    }

    public static int idade(int pAno)
    {
      if (pAno > DateTime.Today.Year)
      {
          throw new Exception($"ano invalido {name
of(pAno)}");
      }
      return DateTime.Today.Year - pAno;
    }
```

```
    }
}
```

Salve e execute o código e execute. O resultado do primeiro *WriteLine* é a expressão completa sem erro. Já o segundo retorna "*ano inválido pAno*", ou seja, agora sabemos qual parâmetro está com erro. Isto é fundamental para ajudar o desenvolvedor e o usuário visualizar uma mensagem explícita, informando exatamente o que deve ser corrigido.

Sendo assim, o uso do *nameof* deverá ser uma obrigação nos códigos com parâmetros. Outra vantagem é que se você renomear o parâmetro *pAno* em todos os lugares onde é referenciado, o *nameof* já captura o parâmetro de qualquer forma.

***Resultado:***

**o peixe tem 11 anos**
**Exceção gerada: 'System.Exception' em livrocsharp.dll**
**ano inválido pAno**

Para efeito didático, vamos criar o método PescaAutorizada que recebe dois parâmetros, o mês e o dia, ambos do tipo *int*. Note as críticas no código, sendo que a pesca só é autorizada se o dia for menor que 16 e o mês estiver entre 5 e 8. Caso alguma destas regras forem aplicadas, o retorno será uma mensagem com o *nameof*, caso contrário, o retorno será a mensagem "Pesca autorizada".

Já no SVM, o bloco do *try..catch* que chama este método há 3 linhas do *WriteLine* para testarmos, sendo que se ocorrer erro cairá no *catch*, então, os dois últimos *WriteLine* devemos comentar ou não, para efeito de testes é claro.

```
using System;
using static System.Console;

namespace livrocsharp
```

```
  {
    class usoDoNameof
    {
      static void Main()
      {
        try
        ...

        // método com 2 parâmetros
        try
        {
          string produto = "peixe";
           DateTime lancto = new DateTime( 2010, 2, 1
8 );
           WriteLine($"o {produto} tem {PescaAutorizad
a(lancto.Month, lancto.Day)}");
            //WriteLine($"o {produto} tem {PescaAutoriz
ada(10, 12)}");
           WriteLine($"o {produto} tem {PescaAutorizad
a(7, 18)}");
        }
        catch (Exception ex)
        {
           WriteLine(ex.Message);
        }
      }

      public static int idade(int pAno)
      ...

       public static string PescaAutorizada(int pMes, in
t pDia)
      {
        if (pDia < 16) {
           throw new Exception($"primeira quinzena {n
ameof(pDia)}");
        }
```

```
        if (pMes >= 5 && pMes <= 8)
        {
            throw new Exception($"pesca no inverno {n
ameof(pMes)}");
        }
        return "Pesca autorizada";
      }
    }
  }
```

Salve e execute novamente para visualizar o resultado customizado com o *nameof*. Sendo assim, com o uso do *nameof* conseguimos identificar os parâmetros, pois não adianta o erro retornar genérico ou o valor em si, nada disso ajuda em saber a exatidão do erro.

***Resultado:***

**o peixe tem 11 anos**
**Exceção gerada: 'System.Exception' em livrocsharp.dll**
**ano inválido pAno**
**o peixe tem Pesca autorizada**
**Exceção gerada: 'System.Exception' em livrocsharp.dll**
**pesca no inverno pMes**

## PROPAGAÇÃO DE NULO E INTERPOLAÇÃO DE STRING

O uso de objetos no .NET nos ajuda muito quando temos dados preenchidos. Há casos em que o objeto ou variável pode estar nulo, e para isto precisamos checar, via código, o conteúdo para dar seguimento à lógica ou validação. Imagine um cadastro de pessoas que aceite nulos! Isto é inaceitável, assim como desviar o fluxo da aplicação para realizar diversos cálculos sendo que o dado ou coleção está nulo.

Neste exemplo, vamos criar os objetos médicos e pacientes,

associar um conjunto de dados e verificar se há informações, por exemplo, será que um médico tem pacientes? Qual o faturamento da clínica se não houve atendimento?

A primeira coisa é criamos um arquivo chamado modeloDados.cs contendo duas classes, Medico e Paciente, respectivamente. Note que um médico pode ter uma coleção de pacientes, então através do *using System.Collections.Generic* podemos usar o *List<Paciente>.* No capítulo sobre OOP tivemos uma lição completa sobre objetos.

A classe Medico tem 3 propriedades, o ID, o Nome e a coleção de Pacientes. A classe paciente tem as propriedades ID, o Nome do paciente, o Histórico (note que o default é o texto "nada relatado", os Remédios, o Valor da consulta (valor default = 0) e, ao final se está Internado.

```
using System.Collections.Generic;

namespace livrocsharp
{
    public class Medico
    {
        public int ID { get; set; }
        public string Nome { get; set; }
        // um médico tem uma coleção de Pacientes
List<T>
        public List<Paciente> Pacientes { get; set; }
    }

    public class Paciente
    {
        public int ID { get; set; }
        public string NomePaciente { get; set; }
         public string Historico { get; set; } = "nada relat
ado";
        public string Remedios { get; set; }
        public decimal ValorConsulta { get; set; } = 0;
```

```
        public bool Internado { get; set; }
    }
}
```

Agora precisamos criar um código para consumir estes objetos. Na mesma pasta, adicione um novo arquivo chamado testarNulos.cs contendo o seguinte código. O objeto médico está contigo na variável med, tendo apenas o ID e o Nome do médico. Este objeto não tem nenhum paciente associado a ele, portanto, está nulo. Ou seja, a propriedade Pacientes (que é uma lista *List<T>)* está vazia.

Se usarmos o *WriteLine* para mostrar os dados do médico com a quantidade de pacientes teremos a sintaxe a seguir. E como contar quantos pacientes tem este médico? Muitas vezes usamos uma variável para contar os pacientes, em seguida usamos um *IF* para saber se é maior que zero, enfim, há muitas formas de fazer tal contagem.

Neste exemplo, usamos uma sintaxe simples e ao mesmo tempo fantástica. Com apenas um caracter (? – interrogação) o C# já pergunta se o objeto é nulo. Mas, como isto funciona?

Acompanhe a explicação da sintaxe a seguir: o med é o objeto médico, o qual tem a propriedade Pacientes que é uma lista *(List<T>)* que pode ter ou não pacientes na coleção. O uso do *Count()* retorna a quantidade de pacientes. Caso tenham pacientes, ou seja, maior que zero *(Count())*, o valor do *Count* é mostrado; caso contrário *??* é mostrado zero.

No primeiro interrogação Pacientes? o C# já pergunta se é nulo ou não (que é justamente o retorno do *Count*). O segundo interrogação (neste caso 2 *??)* é uma sintaxe do C# que já verifica se há ou não conteúdo, neste caso, se existem pacientes ou não. Se não tiverem pacientes, mostra zero (0); caso contrário, mostra a quantidade de pacientes da coleção.

```
med.Pacientes?.Count() ?? 0
```

Isto também poderia ter sido escrito com códigos checando com *IF* se a coleção está nulo ou não (*Lenght*), ou maior que zero, enfim, como disse, há várias maneiras.

```
using static System.Console;
using System;
using System.Collections.Generic;
using System.Linq;

namespace livrocsharp
{
    class testarNulos
    {
        static void Main(string[] args)
        {
            // objeto Medico
             var med = new Medico() { ID = 1, Nome = "Ro
drigo" };

            // quantos pacientes tem este médico?
                WriteLine($"Dr(a) {med.Nome} tem {med.
Pacientes?.Count() ?? 0} pacientes");
        }
    }
}
```

Salve o arquivo, abra o .csproj para definir o testarNulos no *StartupObject* e execute F5.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
```

```
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
            <StartupObject>livrocsharp.testarNulos</
StartupObject>
   </PropertyGroup>
  </Project>
```

Veja o resultado completo contendo zero pacientes.

***Resultado:***

**Dr(a) Rodrigo tem 0 pacientes**

Agora vamos a um exemplo completo, onde um médico tem uma coleção de pacientes. A variável *objM* (objeto médico) contém o ID, o nome do médico e os Pacientes. Como é uma coleção, atente na sintaxe *Pacientes = new List<Paciente>.* A cada paciente novo *new Paciente* temos as propriedades atribuídas aos valores. E, nem todos os pacientes tem dados, por exemplo, remédios, valor da consulta e histórico. Lembre-se dos valores *default* para as propriedades Historico e ValorConsulta, caso não sejam atribuídos, assumem o *default*.

```
static void Main(string[] args)
{
   ...

   var objM = new Medico()
   {
      ID = 1,
      Nome = "Alisson",
      Pacientes = new List<Paciente> {
         new Paciente{
            ID=1,
            NomePaciente="Jeu",
            ValorConsulta=150},
         new Paciente{
```

```
                ID=2,
                NomePaciente="Lucimara",
                ValorConsulta=180,
                Remedios = "simeco plus"},
            new Paciente{
                ID=3,
                NomePaciente="Marcos",
                ValorConsulta=180,
                Remedios = "neosaldina"},
            new Paciente{
                ID=4,
                NomePaciente="Rodrigo",
                ValorConsulta=320,
                Internado = true,
                Historico = "pé chato" },
            new Paciente{
                ID=5,
                NomePaciente="Lucas",
                ValorConsulta=100,
                Internado = true,
                Historico = "camisa de força" },
                    }
                };

            // listar todos os pacientes
             // nome do paciente - remédios (sem remédi
os) - historico
   WriteLine($"historico dos pacientes ---- Dr(a) {objM.
Nome}");
   objM.Pacientes?.ForEach(p =>
   WriteLine($"--- {p.NomePaciente} - remédios: {p.Rem
edios ?? "sem remédios"} - histórico: {p.Historico}"));
   }
```

Aqui vale uma explicação detalhada do bloco que lista os dados. No primeiro *WriteLine* o nome do médico é mostrado. Na linha se-

guinte temos a variável *objM.Pacientes?* o qual verifica se há pacientes para este médico. Lembre-se que a propriedade *Pacientes* é uma coleção *List<T>* e o uso do interrogação já verifica se é ou não nulo.

Neste caso, como há pacientes, o *ForEach* dispara um *looping* para executar o *WriteLine* para cada paciente da lista deste médico, mostrando os dados das propriedades.

O código *p.Remedios ?? "sem remédios"* verifica se a propriedade *Remedios* está nula, caso não esteja mostra o remédio em si, caso contrário mostra o texto sem remédios. Veja que sintaxe limpa e fácil de entender.

```
WriteLine($"Histórico dos pacientes ---- Dr(a) {objM.
Nome}");
objM.Pacientes?.ForEach(p =>
WriteLine($"--- {p.NomePaciente}
- remédios: {p.Remedios ?? "sem remédios"}
- histórico: {p.Historico}        - consulta: {p.
ValorConsulta:n2}"));
```

Salve e execute F5. Observe no resultado que os pacientes que não tem as propriedades com valores atribuídos, recebem o valor padrão.

***Resultado:***

**Histórico dos pacientes ---- Dr(a) Alisson**
**--- Jeu - remédios: sem remédios - histórico: nada relatado - consulta: 150,00**
**--- Lucimara - remédios: simeco plus - histórico: nada relatado - consulta: 180,00**
**--- Marcos - remédios: neosaldina - histórico: nada relatado - consulta: 180,00**
**--- Rodrigo - remédios: sem remédios - histórico: pé chato -**

**consulta: 320,00**
**--- Lucas - remédios: sem remédios - histórico: camisa de força - consulta: 100,00**

Já que temos uma coleção de pacientes, como verificar se há dados e responder as seguintes perguntas:

**1 – Qual o faturamento da clínica?**

Se cada paciente tem o valor da consulta, basta somar a propriedade *ValorConsulta*, então usamos o *SUM(v => v.ValorConsulta)*, sendo que a letra v pode ser qualquer letra, pois será criada em tempo de execução. E isto será feito apenas se Pacientes for diferente de nulo *(Pacientes?).*

**2 – Quantos pacientes estão internados?**

Aqui o filtro será pela propriedade *Internado* = *true*, seguido do contador, *Count(i => i.Internado).*

**3 – Qual a quantidade de pacientes que tomam remédios?**

Temos duas sintaxes válidas, ambas contam *Count* e filtram pela propriedade *Remedios*. A primeira usa o *!= null* (diferente de nulo) e a segunda usa o *!String.IsNullOrEmpty(...)* onde ! (negativa) *String.IsNullOrEmpty* (é nulo ou vazio de uma string).

```
// listar todos os pacientes
...

// qual o faturamento da clinica?
WriteLine($"Faturamento: {objM.Pacientes?.Sum(v => v.ValorConsulta):n2}");

// quantos pacientes estão internados?
WriteLine($"Qtde internados: {objM.Pacientes?.Count(i => i.Internado)}");

// quantos pacientes tomam remédios?
WriteLine($"Qtde tomam remédios (!=null):
```

```
{objM.Pacientes?.Count(r => r.Remedios != null)}");
WriteLine($"Qtde tomam remédios (!String):
{objM.Pacientes?.Count(r => !String.IsNullOrEmpty(r.
Remedios))}");
```

Execute F5 e teremos o resultado a seguir. O mais importante é você ter em mente que estas sintaxes devem ser usadas para quaisquer situações, não importa se os dados estão em coleções, listas, bancos de dados, etc.

***Resultado:***

**Faturamento: 930,00**
**Qtde internados: 2**
**Qtde tomam remédios (!=null): 2**
**Qtde tomam remédios (!String): 2**

# CAPÍTULO 10 – PROCESSAMENTO ASSÍNCRONO

Programação assíncrona permite que a sua aplicação responda de maneira mais rápida e com fluidez, mas pode também causar bastante problema e confusão se não for utilizada da maneira correta.

## O QUE É PROGRAMAÇÃO ASSÍNCRONA

Vamos entender alguns conceitos sobre os tipos de programação:

- Síncrono
- Assíncrono
- MultiThreading
- Concorrente
- Paralelismo

**Síncrono:** Quando você executa um código de maneira síncrona, significa que você precisa esperar o código terminar para poder prosseguir.

**Assíncrono:** Neste modelo você não precisa esperar o código terminar para poder prosseguir. É comum neste modelo de programação o uso de "call-backs", que são chamados ao final da operação.

**MultiThreading:** No multithreading você pode executar vários

códigos ao mesmo tempo. Entenda que uma "thread" é uma execução de código e podemos ter dezenas, centenas delas executando neste exato momento em seu computador. O Windows é um sistema multithread, pois consegue realizar várias tarefas em paralelo, como copiar um arquivo enquanto você navega na internet.

**Concorrente:** Significa que temos duas ou mais tarefas sendo executadas em conjunto, não necessariamente ao mesmo tempo, mas juntas.

**Paralelismo:** Neste cenário temos duas ou mais tarefas sendo executadas ao mesmo tempo, por exemplo em um computador com vários núcleos (cores).

## QUANDO DEVEMOS USAR SÍNCRONO OU ASSÍNCRONO

Se você tem processamentos longos, como acesso a dados, rede, leitura e escrita de arquivos, você pode usar programação assíncrona. Códigos com alto uso de processamento também são candidatos a programação assíncrona. Mas se o seu código tem um fluxo contínuo e cada operação precisa sempre aguardar a anterior, você pode usar o modelo síncrono. Vale a pena considerar tornar o código assíncrono sempre que possível, pois isto aumenta a performance.

Um alerta importante: a programação assíncrona usa mais recursos do computador, então é importante dimensionar corretamente a máquina, pois você terá muito mais código sendo executado ao mesmo tempo, e isto também pode aumentar a complexidade na hora de fazer depuração (debug).

## OPERAÇÕES I/O BOUND E CPU BOUND

Quando falamos de processamento assíncrono, temos que levar em consideração dois aspectos:

- I/O Bound – operações que acessam recursos como disco,

rede para realizar uma operação, por exemplo:

```
var web = new HttpClient();
var site = await web.GetStringAsync(new Uri("https://
www.microsoft.com”));
```

Neste exemplo, estamos baixando o conteúdo do site da Microsoft, como um texto, então estamos usando recursos de rede para acessar o site. A palavra “async” é quem permite a execução assíncrona e iremos falar dela mais adiante.

- CPU bound – operações que executam muitos cálculos, que utilizam muito o processador da máquina, por exemplo:

```
var lista = new List<string>();
lista.Add(“Maria”);
lista.Add(“Joao”);
lista.Add(“Antonio”);
lista.Add(“Joaquim”);
lista.Sort();
```

Neste exemplo, estamos ordenando uma lista com Sort(), então estamos usando processamento.

## ASYNC E AWAIT

Para programarmos de maneira assíncrona vamos utilizar sempre dois objetos: *Task* e *Task<T>*, que permitem este tipo de operação. Também utilizaremos as palavras chave *async* e *await*, que é onde tudo acontece. Estas duas palavras-chave são o ponto central da programação assíncrona.

A palavra *await* inicia a execução assíncrona do código, controlando o fluxo de execução e retornando ao ponto da chamada para continuar o fluxo. E a palavra *async* permite o código ser executado de maneira assíncrona.

Para entendermos melhor como funciona o fluxo de execução do async/await, vejamos a Figura a seguir.

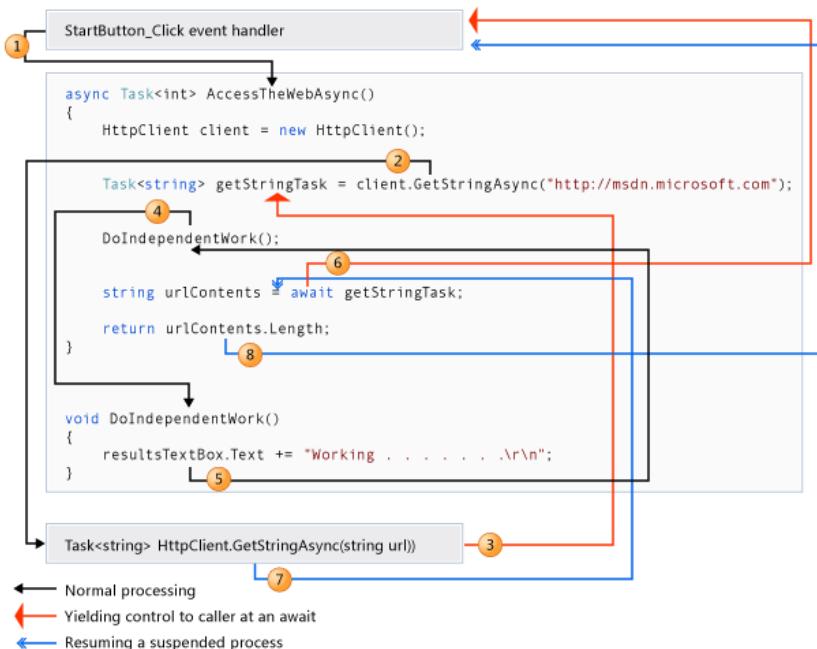



Figura 52: Fluxo do async/await

Vamos entender o fluxo da aplicação:

1. Um evento de botão é disparado e chama um método assíncrono.
2. O método *AccessTheWebAsync()* cria um objeto *HttpClient* e chama *GetStringAsync().*
3. O método GetStringAsync() inicia a execução, mas como ele é assíncrono, o fluxo é retornado para o chamador (AccessTheWebAsync). Ele deve retornar uma string *getstringTask*, então o processo segue para onde a string não é necessária.
4. Como o método *DoIndependentWork* não precisa do *getStringTask*, ele pode ser executado.

5. O médodo *DoIndependentWork()* realiza seu trabalho e retorna.
6. Para preencher a variável *urlContents* é preciso o valor de *getStringTask*, então o fluxo é novamente retornado para o chamador até que o valor seja retornado.
7. *GetStringASync()* processa os dados e retorna então, o controle para o ponto onde o valor era necessário.
8. Finalmente o tamanho da string é calculado e retornado.

O código pode parecer um tanto complicado de entender e de fato, a primeira vista, ele é. Mas a cada novo código assíncrono vamos entendendo melhor o funcionamento. Mas cuidado, apenas colocar a palavra-chave *async* ou *await* não torna o método automaticamente assíncrono, para isto acontecer devemos ter operações assíncronas acontecendo dentro do método, como ilustrado na figura anterior.

Para deixar mais claro o conceito de execução de operações assíncronas,vamos construir um exemplo de acordo com a listagem abaixo, que é muito parecida com o fluxo da figura anterior.

```
   public partial class frmPrincipal : Form
   {
      string site;
      public frmPrincipal()
      {
         InitializeComponent();
          this.btnLer.Click += async (o, e) => { await btnL
er_Click(o, e); };
         site = "";
      }

       private async Task btnLer_Click(object sender, Eve
ntArgs e)
      {
         lstStatus.Items.Add("Lendo site...");
         lstStatus.Refresh();
```

```
            var web = new HttpClient();
            site = await web.GetStringAsync(new Uri(txtSite.
Text));
            if (site != “” && txtPalavra.Text != “”)
            {
                lstStatus.Items.Add(“Site lido”);
                btnContar_Click(sender, e);
            }
        }

         private void btnContar_Click(object sender, Event
Args e)
        {
            if (txtPalavra.Text != “”)
            {
                    var total = Regex.Matches(site, txtPalavra.
Text).Count();
                lstStatus.Items.Add($”Total de ocorrências da
palavra {txtPalavra.Text} = { total}”);
            }
            else
            {
                lstStatus.Items.Add(“Palavra não informada”);
            }
        }
    }
```

Esta é uma aplicação Windows Forms, então temos interação do usuário com a tela (UI) através de dois botões. Na figura  temos a tela da aplicação em execução.

Exemplo de Download Assíncrono

URL https://www.microsoft.com/pt-br/ Ler

Palavra Office Contar

Lendo site...
Total de ocorrências da palavra Office = 0
Site lido
Total de ocorrências da palavra Office = 13

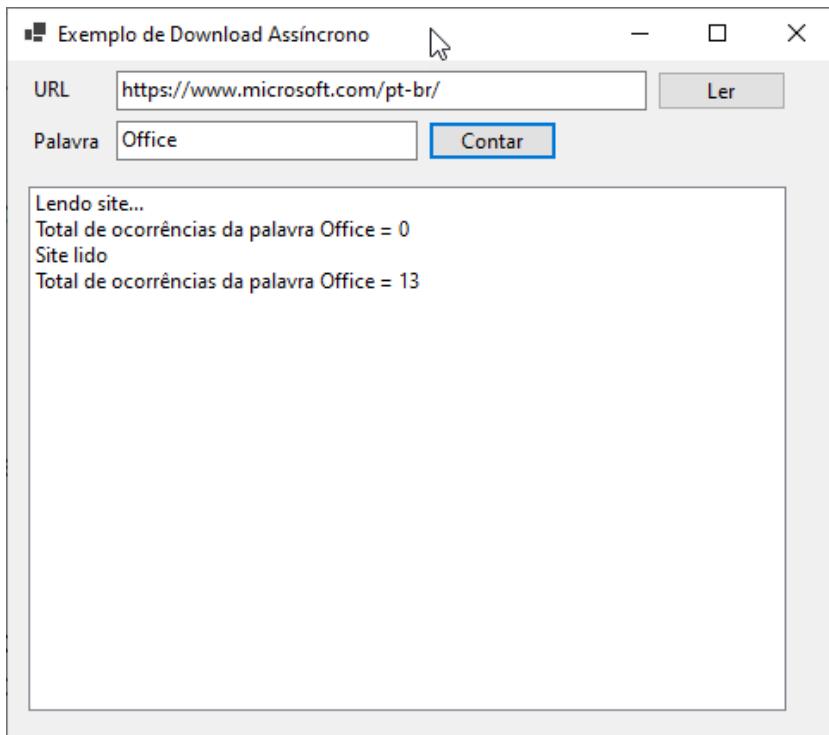



Figura 53: Tela da aplicação DownloadAssincrono

A ideia deste exemplo é baixar o conteúdo do site e permitir que o usuário conte quantas palavras existem no site informado. O exemplo da *figura 2,* temos vários textos ilustrando o que aconteceu durante a execução: quando o usuário clicou no botão "Ler" temos o texto "Lendo site...", mas antes mesmo de terminar a leitura, o usuário clicou no botão "Contar" e, o resultado foi "Total de ocorrências da palavra Office = 0", pois o site ainda não tinha sido lido totalmente, mesmo assim o usuário conseguiu clicar no botão. Isto ilustra o fluxo de execução do processamento assíncrono.

Agora vamos entender o código. Primeiramente temos no construtor do nosso Form o código abaixo:

```
this.btnLer.Click += async (o, e) => { await btnLer_
```

```
Click(o, e); };
   site = "";
```

Este código atribui um evento ao botão *Ler* que chama o método *btnLer_Click()* de forma assíncrona, utilizando a palavra-chave *await.*

Dentro do código do Click temos o seguinte:

```
   private async Task btnLer_Click(object sender, Event
Args e)
   {
     var web = new HttpClient();
      site = await web.GetStringAsync(new Uri(txtSite.
Text));
     if (site != "" && txtPalavra.Text != "")
     {
       lstStatus.Items.Add("Site lido");
       btnContar_Click(sender, e);
     }
   }
```

A definição do método tem o objeto *Task* e palavra-chave *async*, que indicam a execução de modo assíncrona e depois temos o código com a chamada do *GetStringAsync()* usando a palavra-chave *await*. Veja que temos uma condição que testa a variável *site*, que é retornada pelo método assíncrono, então o código do *if()* só será executado quando este valor for retornado. Então, acontece a chamada do botão *Contar*, que faz a contagem de palavras usando a classe *Regex*.

Temos então, para um método assíncrono a definição com *async Task* e dentro do método temos a chamada assíncrona com *await.* Veja que o método btnContar_Click() não é assíncrono e não tem a mesma definição do btnLer_Click().

Na figura a seguir estamos executando o código e colocando um break-point, onde podemos ver como as threads estão sendo exe-

cutadas na memória.

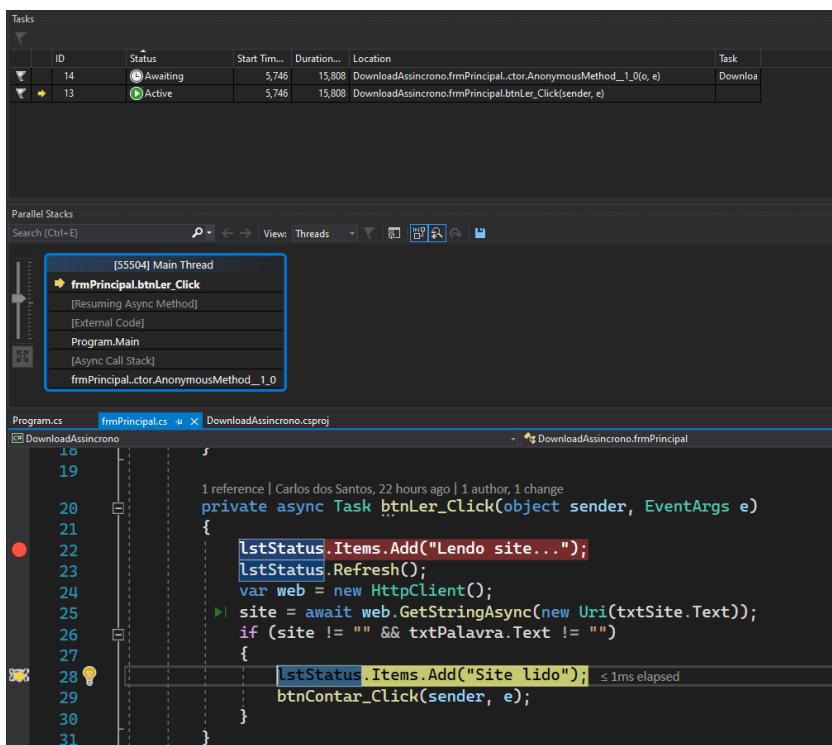



Figura 54. Depuração da aplicação assíncrona.

A figura anterior mostra que temos uma thread aguardando (Awaiting) e outra ativa (Active), e, conforme vamos navegando pelo código, as threads são finalizadas. Experimente debugar o código no seu Visual Studio.

## DECLARAÇÃO TASK NOS MÉTODOS

Um método assíncrono tem em sua definição a palavra-chave *async* e o objeto *Task* que representa o retorno do método. Veja que o *Task* pode ter um tipo de dados para retorno, por exemplo *Task<string>* retornará uma *string*.

Depois de declarado o método com *async Task*, espera-se que ele possua uma chamada para um código assíncrono, através da palavra-chave *await*.

### TASK.RUN()

Em operações CPU Bound, onde temos uso apenas de CPU, podemos criar métodos assíncronos com o método *Task.Run()*. Mas fique atento para usar este método somente em códigos que fazem uso de CPU e não uso de I/O.

Para ilustrar o uso do *Task.Run()* vamos modificar o nosso código anterior e transformar a contagem de palavras em um método assíncrono. Veja a listagem a seguir:

```
private async void btnContar_Click(object sender, EventArgs e)
{
    if (txtPalavra.Text != "")
    {
      var total = await ContarPalavras(txtPalavra.Text);
       lstStatus.Items.Add($"Total de ocorrências da palavra {txtPalavra.Text} = { total}");
        }
    else
    {
      lstStatus.Items.Add("Palavra não informada");
    }
}

private async Task<int> ContarPalavras(string palavra)
{
        return await Task.Run(() => Regex.Matches(site, palavra).Count());
}
```

Agora temos um novo método chamado ContarPalavras() que pode ser executado de forma assíncrona, pois sua definição tem *async Task* e também estamos informando o tipo do dado do retorno como *int*.

O segredo aqui é a execução com *Task.Run()*, onde passamos o código que será executado de forma assíncrona e pegarmos o retorno.

A execução da aplicação ocorre da mesma maneira, mas agora tornamos o código ainda mais responsivo.

## BOAS PRÁTICAS COM ASYNC

Sempre que você criar um método assíncrono, coloque a extensão *Async* no final do nome, pois isto indicará que o método aceita ser executado com *await*.

O código a seguir mostra uma WebAPI em .NET 5.0 com um método síncrono e outro assíncrono:

```
namespace ExemploAPI.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class DadosController : ControllerBase
    {
        private readonly ILogger<DadosController> _logger;

        public DadosController(ILogger<DadosController> logger)
        {
            _logger = logger;
        }

        [HttpGet("sincrono")]
```

```
        public string Get()
        {
           return "Exemplo de API - método sincrono";
        }

        [HttpGet("assincrono")]
        public async Task<string> GetAsync()
        {
            var sr = new StreamReader("./arquivos/teste.
txt");
           var conteudo = await sr.ReadToEndAsync();
           sr.Close();
           return conteudo;

        }
     }
  }
```

Veja que o método *Get()* é síncrono, pois não realiza operações assíncronas, mas o método *GetAsync()* faz a leitura de um arquivo de forma assíncrona (ReadToEndAsync), e por isto, tem a declaração *async Task<string>*, pois o retorno é assíncrono, do tipo string.

A programação assíncrona pode ser aplicada a diversos tipos de aplicação, como no exemplo anterior, que é uma aplicação web.

O uso de programação assíncrona pode criar um ótimo benefício para o seu código, mas lembre-se de se aprofundar e ficar atento às regras que demonstramos aqui.

# CAPÍTULO 11 – COMPONENTIZAÇÃO

No mundo do desenvolvimento de softwares, temos muitas bibliotecas disponíveis para adicionarmos aos projetos e usá-las. Muitas são Open-source, basta fazer download e usar, outras são pagas (componentes de interface de usuário), e outras são desenvolvidas dentro da equipe com funcionalidades bem específicas.

Todo projeto de biblioteca gera uma extensão DLL, o que significa que se pegarmos uma DLL criada em .NET, podemos usá-la em qualquer projeto .NET. Via de regra sim, mas depende das versões e dos tipos de projetos.

A vantagem principal de componentizar é a reutilização de determinada funcionalidade, por exemplo, validação de usuário, rotinas de impressão, gerar QR Code, calcular um imposto, enfim, alguma rotina que podemos escalar. Como elas serão referenciadas nos projetos, quando houver alguma nova versão da DLL, basta distribuir ou atualizar a referência para que sejam refletidas as alterações.

Se pensarmos que o .NET Core tudo é adicionado conforme a necessidade, usando o conceito de injeção de dependência (DI), podemos afirmar que o .NET Core é componentizado. Basta ver a quantidade enorme de bibliotecas ou pacotes que estão disponíveis no site https://www.nuget.org.

Para este livro iremos criar um projeto de Class Library (DLL), o qual será referenciado e usado no projeto de Console. Para isto,

crie um novo diretório no seu computador chamado, por exemplo, ComponentesDLLs. No meu está em C:\Treinamentos\CS\ComponentesDLLs. Como estamos usando o VS Code, abra-o, selecione o menu *Terminal* / New Terminal. Para criar o projeto de DLL, no *Terminal*, digite o comando a seguir:

```
dotnet new classlib --name CompDLL
```

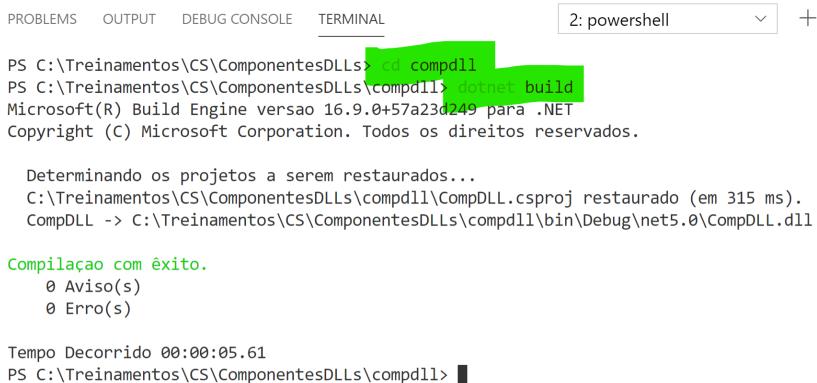
O *dotnet new* informa que será um novo projeto, o tipo de projeto será *classlib (Class Library / DLL)*, o nome do projeto será *CompDLL*. Pressione *ENTER* e aguarde a criação.

Na lista de arquivos, abra o *CompDLL.csproj* para ter certeza que o framework será o netstandard2.1. Caso esteja como net5.0, basta trocar para netstandard2.1.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
                        <TargetFramework>netstandard2.1</
TargetFramework>
  </PropertyGroup>
</Project>
```

Automaticamente é criado o arquivo Class1.cs, pode renomear para CalculoGeral.cs. No nome da classe também pode renomear para este mesmo nome. Veja no código a seguir que a classe CalculoGeral contém duas propriedades (valor e dias) e o método FaturamentoMedioDiario, cujo resultado é a divisão do valor do faturamento pela quantidade de dias. O importante é definir que a classe terá visibilidade pública (public class), não esqueça disso.

```
using System;

namespace CompDLL
```

```
  {
      public class CalculoGeral
      {
          public int valor { get; set; }
          public int dias { get; set; }

           public int? FaturamentoMedioDiario() => valor
/ dias;
      }
  }
```

Salve o arquivo. Como este projeto é uma DLL e não um executável, não adianta pressionar F5, precisamos compilar através da linha de comando. Então, no Terminal, digite o comando "cd compdll" para ir na pasta do projeto, neste caso, compdll (CompDLL está contigo na pasta ComponentesDLLs). E, para compilar, digite "dotnet build".

Conforme a figura a seguir, o resultado da compilação foi com sucesso, gerando a DLL física CompDLL.dll na pasta C:\Treinamentos\CS\ComponentesDLLs\CompDLL\bin\Debug\net5.0.

Figura 55 – Build do projeto DLL

Pronto, basta referenciar em outros projetos e usa-la. Claro que estamos num ambiente de Debug, mas no momento em que a DLL estiver testada e finalizada, basta usar o comando a seguir para gerar a DLL no modo Release para ser distribuída.

```
dotnet build --configuration Release
```

## COMO REFERENCIAR OUTRO CSPROJ NO PROJETO ATUAL?

Agora que a DLL está compilada, abra o projeto livroCsharp no VS Code. Abra o Terminal e digite o comando a seguir para referenciar o projeto de componente CompDLL.csproj. Para isto, use o *dotnet add reference<caminho completo csproj>*.

```
dotnet add reference C:\Treinamentos\CS\ComponentesDLLs\CompDLL\CompDLL.csproj
```

Caso queira saber quais projetos estão sendo referenciados, use:

```
dotnet list reference
```

Se você abrir o arquivo livrocsharp.csproj verá que há a referência ao CompDll.

```
<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
     <ProjectReference Include="..\ComponentesDLLs\CompDLL\CompDLL.csproj" />
  </ItemGroup>

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
          <StartupObject>livrocsharp.testarNulos</StartupObject>
```

```
                    <GenerateAssemblyInfo>false</
GenerateAssemblyInfo>
    <GenerateTargetFrameworkAttribute>false</Gener
ateTargetFrameworkAttribute>
   </PropertyGroup>
  </Project>
```

## COMO CONSUMIR A DLL?

Agora com o projeto de CompDLL referenciado, cria uma nova pasta chamada Cap11 e o arquivo ConsomeDLL.cs, conforme código a seguir:

```
using static System.Console;
using System;
using CompDLL;

namespace livrocsharp
{
    class ConsomeDLL
    {
        static void Main(string[] args)
        {
            int valorFat = 100000;
            int diasFat = 50;
            CalculoGeral objFat = new CalculoGeral()
                    { valor = valorFat, dias = diasFa
t };
                    WriteLine($"Faturamento: {ob
jFat.valor:n0} - dias {objFat.dias} - Média: {objFat.
FaturamentoMedioDiario():n0}");
        }
    }
}
```

Antes de mais nada é preciso declarar o using CompDLL para termos acesso à classe. No SVM (*static void Main*) declaramos duas

variáveis do tipo *int* sendo valorFat (valor do faturamento) e diasFat (dias faturados).

Em seguida, temos a classe CalculoGeral atribuída ao *objFat* passando os conteúdos das propriedades valor e dias. No *WriteLine* os dados são mostrados e o método FaturamentoMedioDiario mostrará o cálculo conforme a fórmula do componente, que divide o valor pelos dias, resultando no faturamento médio diário.

Salve e execute F5 para ver o resultado.

***Resultado:***

**Faturamento: 100.000 - dias 50 - Média: 2.000**

## MÉTODO COM ENUM E SWITCH

Vamos melhorar o nosso componente para calcular o valor do seguro do carro de acordo com a cor. No projeto do Componente CompDLL, adicione a classe DescontoGeral contendo um enumerador (enum Cores) com 3 cores que serão usadas para calcular o desconto e o método DescontoPorCor.

O enumerador é uma forma de deixar a leitura do código mais clara através de expressões textuais, os quais podem representar um número, como o nosso código.

Já o método DescontoPor Cor é estático (acessível direto pelo nome da classe), recebe dois parâmetros, o valor do seguro e a cor a ser avaliada para o desconto, neste caso, o próprio texto do enumerador. Aqui usamos uma sintaxe moderna do switch, o qual é declarada a variável cor seguida da palavra chave switch. Dentro do bloco temos todas as opções de cores, e dependendo da cor é aplicado o desconto de 90, 80 ou 70%. Note que a sintaxe verifica qual a cor e se for esta, o sinal de => atribui o cálculo a ser processado, neste caso, o valor do seguro vezes o desconto.

O valor padrão nesta sintaxe DEVE ficar na última opção e é representado pelo caractere (_), neste caso, não tem desconto.

```
using System;

namespace CompDLL
{
   public class CalculoGeral
   ...

   public class DescontoGeral
   {
      public enum Cores
      {
         Vermelho = 1,
         Verde = 2,
         Azul = 3
      }

       public static double DescontoPorCor(double valor, Cores cor) =>
      cor switch
      {
         Cores.Vermelho => valor * 0.9,
         Cores.Verde => valor * 0.8,
         Cores.Azul => valor * 0.7,
         _ => valor,
      };
   }
}
```

Salve e compile o projeto. Retorne para o arquivo ConsomeDLL.cs no projeto livrocsharp. Adicione os códigos a seguir para com as variáveis declaradas do valorSeguro, corVerde, corAzul e corVermelho. Cada uma destas cores refereciam o enumerador DescontoGeral.Cores.Cor.

Em seguida, o WriteLine mostra as mensagens contendo o valor bruto e o valor com desconto de acordo com a cor.

```
using static System.Console;
using System;
using CompDLL;

namespace livrocsharp
{
    class ConsomeDLL
    {
        static void Main(string[] args)
        {
            int valorFat = 100000;
            …

            // acessa o desconto do seguro por cor
            double valorSeguro = 2500;
            var corVerde = DescontoGeral.Cores.Verde;
            var corAzul = DescontoGeral.Cores.Azul;
                var corVermelho = DescontoGeral.Cores.
Vermelho;
            WriteLine("-------- DESCONTO POR CORES -----
-");
            WriteLine($"R$ bruto: {valorSeguro:n0} - des
conto {corAzul}: {DescontoGeral.DescontoPorCor(valorS
eguro, corAzul):n0}");

            WriteLine($"R$ bruto: {valorSeguro:n0} - des
conto {corVerde}: {DescontoGeral.DescontoPorCor(valo
rSeguro, corVerde):n0}");

            WriteLine($"R$ bruto: {valorSeguro:n0} - des
conto {corVermelho}: {DescontoGeral.DescontoPorCor(v
alorSeguro, corVermelho):n0}");
        }
    }
```

```
}
```

Salve e execute F5. Note que quando executamos este projeto, todos os projetos adicionados à este também são compilados, ou seja, o projeto CompDLL é compilado antes deste livrocsharp.

***Resultado:***

**—— DESCONTO POR CORES ——**
**R$ bruto: 2.500 - desconto Azul: 1.750**
**R$ bruto: 2.500 - desconto Verde: 2.000**
**R$ bruto: 2.500 - desconto Vermelho: 2.250**

# CAPÍTULO 12 – INSTALAÇÃO / DEPLOY DO PROJETO

Fazer a instalação de uma aplicação ou fazer o deploy (implantar) como normalmente falamos no dia a dia, é a parte mais emocionante de um projeto, pois materializa todo o esforço e trabalho da construção do software. Mas a instalação requer também muitos cuidados que precisam ser tomados durante todo o ciclo de desenvolvimento, como por exemplo: o que é preciso para executar a aplicação: qual hardware, quais softwares, o que precisamos colocar no ambiente de produção?

Implantar o sistema significa gerar os arquivos binários (EXE, DLL) e demais arquivos que compõem a nossa aplicação e colocá-los no ambiente onde serão executados. Uma publicação pode conter dezenas, às vezes, centenas de arquivos.

## AMBIENTES PARA EXECUÇÃO DA APLICAÇÃO

Quando falamos ambientes (no plural) significa que podemos ter mais de um ambiente, como por exemplo: Desenvolvimento, Homologação e Produção. É claro que isto pode variar dependendo do projeto, da equipe e da empresa, podemos ter mais ou menos ambientes. Mas vamos considerar estes 3 inicialmente:

- Desenvolvimento: onde trabalhamos diariamente, ou seja, onde a aplicação tem o código mais atualizado
- Homologação: onde colocamos um código já finalizado, para

que sejam feitos testes que garantam a qualidade da aplicação. Estes testes já podem ter sido feitos de maneira automática (Testes Unitários) no ambiente de desenvolvimento, mas aqui, na homologação, iremos simular o uso da aplicação
- Produção: este é o destino do nosso código, onde a aplicação irá atender aos usuários, onde ela de fato vai rodar.

## MODOS DE COMPILAÇÃO

Os arquivos binários são gerados durante o processo de compilação da nossa aplicação e antes de instalarmos estes binários nos ambientes, é importante entender com eles devem ser gerados e, para isto, existem dois modos de compilação: Debug e Release.

### Modo de Debug:

Em Debug, nossa aplicação permite ser executada linha a linha, onde podemos depurar, ou executar o código linha por linha, parte por parte, a fim de resolvermos problemas e garantirmos que tudo está funcionando.

Neste modo, o Visual Studio, gera um arquivo extra, com a extensão .PDB (Program Database), que inclui informações sobre o código fonte. Isto permite também que façamos depuração em um ambiente sem o código fonte. Por exemplo, você tem uma DLL dentro do seu projeto e ao executar o Debug, acessando as linhas do seu código, se depara com um código externo, dentro da DLL. Se esta DLL tiver o arquivo PDB, será possível navegar pelo código fonte que ela possui. O modo Debug é utilizado geralmente durante o desenvolvimento.

### Modo Release:

Este é o modo que utilizamos para colocar nosso código em produção, pois ele é mais otimizado pelo compilador, ou seja, não contém informações para Debug, somente execução. Sempre que for

instalar seu software, compile no modo Release.

Dentro do Visual Studio, você pode escolher o modo de compilação no menu superior, conforme a figura a seguir:

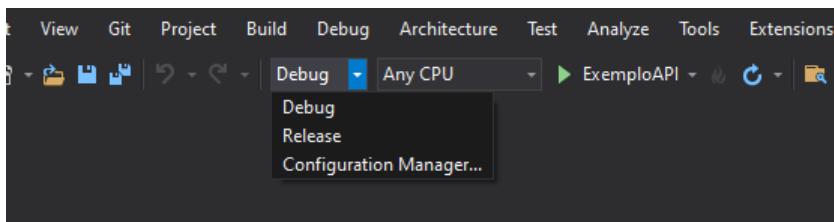



Figura 56 - Modo Debug/Release no Visual Studio

Se você gosta da linha de comando e está usando .NET Core, é possível fazer a mesma coisa com os comandos a seguir:

```
dotnet publish -o publicacao
dotnet publish -c release -o publicacao
```

O primeiro comando gera os binários no diretório publicação, em modo debug e o segundo em modo release.

O Visual Studio cria duas pastas durante o processo de desenvolvimento, dentro do diretório do seu projeto: bin/Debug e bin/Release. O “bin” é de binário e a outra pasta se refere a como você compilou a aplicação. Veja na figura a seguir:

PC › Dados (D:) › LivroCS › ExemploAPI › bin

| Name | Date modified | Type | Size |
|---|---|---|---|
| Debug | 31/05/2021 21:13 | File folder | |
| Release | 31/05/2021 21:29 | File folder | |

Figura 57 - Pastas Debug e Release

Agora olhando dentro da pasta Release, temos os arquivos da figura a seguir:

is PC > Dados (D:) > LivroCS > ExemploAPI > bin > Release > net5.0 >

| Name | Date modified | Type | Size |
|---|---|---|---|
| ref | 31/05/2021 21:29 | File folder | |
| appsettings.Development.json | 31/05/2021 21:13 | JSON File | 1 KB |
| appsettings.json | 31/05/2021 21:13 | JSON File | 1 KB |
| ExemploAPI.deps.json | 31/05/2021 21:29 | JSON File | 109 KB |
| ExemploAPI.dll | 31/05/2021 21:29 | Application exten... | 10 KB |
| ExemploAPI.exe | 31/05/2021 21:29 | Application | 125 KB |
| ExemploAPI.pdb | 31/05/2021 21:29 | Program Debug D... | 20 KB |
| ExemploAPI.runtimeconfig.dev.json | 31/05/2021 21:29 | JSON File | 1 KB |
| ExemploAPI.runtimeconfig.json | 31/05/2021 21:29 | JSON File | 1 KB |
| Microsoft.OpenApi.dll | 29/08/2020 17:14 | Application exten... | 170 KB |
| Swashbuckle.AspNetCore.Swagger.dll | 23/09/2020 09:47 | Application exten... | 16 KB |
| Swashbuckle.AspNetCore.SwaggerGen.dll | 23/09/2020 09:47 | Application exten... | 79 KB |
| Swashbuckle.AspNetCore.SwaggerUI.dll | 23/09/2020 09:47 | Application exten... | 3.231 KB |

Figura 58 - Arquivo da Pasta Release

No exemplo acima, temos uma aplicação ASP.NET Core WebAPI, veja que ela tem vários arquivos: dll, json, exe. Todos estes arquivos serão copiados para o ambiente onde nossa aplicação será executada.

## DISTRIBUINDO NOSSA APLICAÇÃO

Como vimos acima, para instalar nossa aplicação, basta gerar uma versão RELEASE do nosso código e depois copiar o conteúdo desta para o ambiente de execução.

Mas para executar a aplicação precisamos de um componente essencial, o **Runtime**.

### Runtime de Execução

Uma aplicação em .NET (atualmente) produz um binário que precisa de um runtime para ser executado, isto porque o nosso binário, contém apenas o nosso código compilado, mas uma aplicação é composta de muito código escrito pela própria Microsoft, e

este código é distribuído através dos runtimes.

O runtime ou JIT (Just in Time) pega o nosso binário e o transforma em linguagem de máquina que pode ser executado pelo sistema operacional. Isto nos permite ter códigos híbridos, ou seja, um mesmo binário pode executar no Windows e no Linux. Mas atenção, prefira sempre produzir os binários para o sistema operacional onde ele será executado, pois isto garante maior otimização e velocidade na execução. O runtime pode ser baixado no site https://dotnet.microsoft.com/download e atualmente possui versões para Windows, Linux, MAC e ARM, quando você está desenvolvendo com .NET Core.

É importante que você instale o runtime adequado para a versão de projeto que você criou, ou seja, se está desenvolvendo para .NET Core 3.1, precisa ter o runtime desta versão, se está desenvolvendo para .NET 5.0 mesma coisa a assim por diante.

## MEIOS PARA INSTALAÇÃO DA APLICAÇÃO

Apesar de parecer simples, copiar os arquivos produzidos pelo Visual Studio na pasta do servidor onde será executado, pode causar sérios problemas como por exemplo mandarmos um código com problema para a produção.

Para nos ajudar com isto, temos ferramentas de DevOps (Desenvolvimento + Operação), que faz a automação deste processo para nós, ou seja, compila o código e coloca no ambiente apropriado de forma automática. O deploy automatizado garante consistência no processo de implantação, evitando que por exemplo esqueçamos algum arquivo para trás, parando o ambiente.

Sempre que possível prefira fazer a instalação através de uma ferramenta automatizada, como Azure DevOps ou GitHub com Actions.

## EXECUTANDO A APLICAÇÃO

A execução de uma aplicação vai depender do tipo desta aplicação, então vamos listar alguns tipos mais comum e como elas são executadas:

- Aplicação Console: precisamos apenas chamar o arquivo .EXE
- Aplicação Desktop: mesmo processo do console, basta chamar o .EXE
- Aplicação Web:
    - Em servidores Windows: você instala o ASP.NET Hosting Bundle (download em https://dotnet.microsoft.com/download) e depois configura a aplicação no Internet Information Server (IIS)
    - Você tambem pode executar fora do IIS usando o um servidor chamado Kestrel, presente no ASP.NET Core, para isto basta chamar: dotnet <DLL da aplicação>  ou apenas o .EXE
- Aplicação como Serviço:
    - Você pode construir serviços para Windows ou Linux, neste caso quem chama a aplicação é o serviço, e você precisa apenas instalar sua aplicação. Para isto existem muitas maneiras de instalar e configurar. Consulte a documentação para o serviço em https://docs.microsoft.com

No nosso exemplo, foi construido uma aplicação ASP.NET Core. Para executar de dentro do Visual Studio, basta pressionar F5, e na linha de comando, podemos usar:

```
dotnet run
```

ou simplesmente o nome do executável da aplicação, no nosso caso ***ExemploAPI.EXE***

O resultado é o mesmo nos dois casos e é mostrado na figura a seguir.

```
❯ dotnet run
Building ...
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: D:\LivroCS\ExemploAPI
```

Figura 59 - Aplicação Web sendo executada

Instalar o software significa pegar os nossos binários e copiar no ambiente onde iremos executá-los, e instalar a versão apropriada do runtime. Lembre-se que manter um ambiente de desenvolvimento separado da produção é importante para o projeto, pois evita que problemas ainda não tratados pelo desenvolvedor vá para o ambiente dos usuários.

Fazer a instalação de uma aplicação ou fazer o deploy (implantar) como normalmente falamos.

# CAPÍTULO 13 – TESTES DE UNIDADE

Quando compramos um produto esperamos que o mesmo funcione corretamente e que tenha uma durabilidade adequada, ou seja, queremos produtos com qualidade. Para atingir esta qualidade, a indústria testa seus produtos extensivamente e com base nos resultados faz as alterações necessárias para que de iteração a iteração, produza produtos cada vez melhores.

Com software não é diferente. Como desenvolvedores, temos o dever de produzir código performático e de qualidade para satisfazer nossos clientes. Os testes são a nossa principal arma para evitarmos que um usuário encontre o famoso "bug" e existem muitas formas de se testar software. Neste capítulo, vamos aprender um pouco sobre um tipo específico de testes, chamado teste de unidade.

## O QUE SÃO TESTES DE UNIDADE?

Os Testes de Unidade são rotinas de software que escrevemos para testar uma parte bem pequena de nosso sistema. Isso mesmo, escrevemos código para testar nosso código. Quanto menor o escopo de um teste, melhor. O teste unitário então não tem o dever de testar seu sistema inteiro, mas sim um único aspecto de seu programa, por isso o nome "unitário". O conjunto de todos os testes unitários que você escrever vai acabar cobrindo boa parte do seu sistema, e quanto mais melhor.

Vamos supor que você tenha uma função super complexa que calcule o desconto dado a um produto. Sabendo de antemão as regras deste desconto, você pode criar um teste que chame sua função de descontos e compare o resultado retornado com um resultado pré-definido por você, te avisando se der errado. Desta forma, você programa já sabendo se sua função vai funcionar ou não e, quando você alterar essa funcionalidade no futuro com mais um caso diferente, pode rodar o seu teste e saber se você não quebrou nada sem querer.

## COMO CRIAR UM TESTE DE UNIDADE

Lendo a seção anterior, você deve ter pensado: mas onde coloco esses testes? Como faço pra rodá-los quando eu precisar? Vou ter que escrever mais um programa pra isso? Não se preocupe, existem muitas ferramentas de testes para C# e podemos facilmente referenciar uma delas (lembra do capítulo sobre componentização?) e utilizá-la em nossos projetos facilitando nossa vida.

Para este livro, usaremos uma ferramenta chamada “xUnit.net”, umas das mais utilizadas na plataforma .Net. O site com informações completas do xUnit encontra-se aqui: https://xunit.net/

## TESTES DE UNIDADE NA PRÁTICA

Criando o projeto principal, escolha um diretório base vazio para sua solução. No meu caso vou colocar em c:\dev\csharp-com-testes, mas você pode escolher qualquer outro lugar. Digite o seguinte comando para criar nosso projeto chamado Loja:

```
dotnet new console -o Loja
```

Você deve receber uma resposta similar a esta:

```
C:\dev\csharp-com-testes>dotnet new console -o Loja
The template “Console Application” was created successfully.
```

```
Processing post-creation actions...
Running 'dotnet restore' on Loja\Loja.csproj...
  Determining projects to restore...
  Restored C:\dev\csharp-com-testes\Loja\Loja.csproj (in 181 ms).
Restore succeeded.
```

Você deve reconhecer o tipo de projeto “console”, pois já o usamos em capítulos anteriores. O argumento “-o Loja” está dizendo para colocarmos este projeto em um subdiretório chamado Loja, assim como para criar nosso projeto com o mesmo nome.

Este será nosso projeto principal. O código que escreveremos para nossos testes será colocado em um projeto separado, de forma a organizar o que vai rodar em produção e o que é código utilizado somente para testar o projeto principal.

**Criando o Projeto de Testes**

Agora vamos criar nosso projeto de testes, digite no mesmo diretório base:

```
dotnet new xunit -o Loja.Tests
```

O resultado deve ser:

```
C:\dev\csharp-com-testes>dotnet new xunit -o Loja.Tests
The template “xUnit Test Project” was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on Loja.Tests\Loja.Tests.csproj...
  Determining projects to restore...
Restored C:\dev\csharp-com-testes\Loja.Tests\Loja.Tests.
csproj (in 1.89 sec).
Restore succeeded.
```

Desta vez, estamos escolhendo o tipo de projeto chamado “xu-

nit", que é um projeto de testes. Para nossa conveniência, este tipo de projeto já vem configurado quando você instala o dotnet.

Não é obrigatório, mas é uma prática comum dar ao seu projeto de testes o mesmo nome do projeto a ser testado, mas com o sufixo ".Tests". Desta forma, qualquer programador .Net vai reconhecer instantaneamente este tipo de projeto só pelo nome sem ter que explorar seu interior.

**Vinculando os dois Projetos**

Neste momento temos 2 projetos: o principal e o de testes. Para que nosso projeto de testes possa testar as funcionalidades do principal, temos que adicionar uma referência entre os dois. O comando para realizar esta operação é o seguinte:

```
dotnet add Loja.Tests\Loja.Tests.csproj reference Loja\Loja.csproj
```

Caso tenha algum problema, verifique se está digitando em seu diretório base e se os nomes dos projetos estão corretos. A resposta deve ser a seguinte:

```
C:\dev\csharp-com-testes>dotnet add Loja.Tests\Loja.Tests.csproj reference Loja\Loja.csproj
Reference `..\Loja\Loja.csproj` added to the project.
```

Pronto, nosso projeto de testes agora consegue "enxergar" funções criadas no projeto principal e isso é exatamente o que precisamos.

Agora vamos criar uma "Solução" que englobe os 2 projetos para que fique mais fácil trabalhar com eles. O comando para criar a solução é o seguinte:

```
dotnet new sln -n Loja
```

Neste caso, vamos criar uma Solução com o nome de Loja. Se não usarmos o argumento "-n" o nome da Solution seria o nome do diretório em que nos encontramos, no caso, "csharp-com-testes", então não esqueça dele.

Após criar a Solução, vamos adicionar nossos 2 projetos a ela:

```
C:\dev\csharp-com-testes>dotnet sln add Loja\Loja.csproj
Project `Loja\Loja.csproj` added to the solution.

C:\dev\csharp-com-testes>dotnet sln add Loja.Tests\Loja.Tests.csproj
Project `Loja.Tests\Loja.Tests.csproj` added to the solution.
```

Pronto, tudo configurado.

**Entendendo a Solução no VS Code**

Abra sua solução no VS Code. Uma forma rápida de fazer isso é digitar no terminal mesmo:

```
C:\dev\csharp-com-testes> code .
```

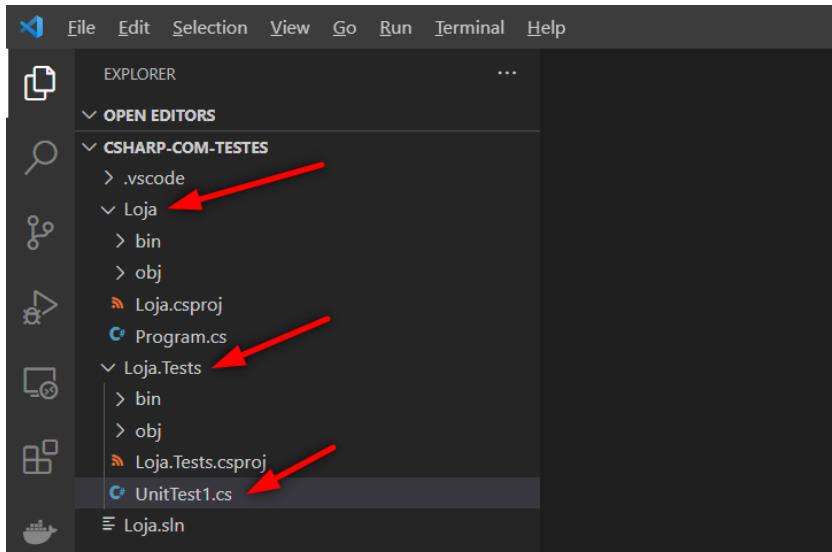



Figura 60 - Estrutura do projeto

Aqui vemos nossos 2 projetos: Loja e Loja.Tests. O projeto de testes já apresenta uma primeira classe chamada UnitTest1, servindo como exemplo de como a estrutura de uma classe de testes pode ser. Vamos dar uma olhada nela:

```
public class UnitTest1
{
    [Fact]
    public void Test1()
    {

    }
}
```

Para o nunit um teste de unidade é qualquer função que tenha o atributo “[Fact]” em cima dela. Você organiza testes sobre uma mesma funcionalidade em classes e é comum ter uma classe de testes para cada classe que você está testando no projeto principal.

O xunit se encarrega de dizer ao dotnet que este é um projeto de testes e, desta forma, você pode utilizar o próprio framework para rodar os mesmos.

## Escrevendo seus primeiros Testes de Unidade

Vamos começar com um exemplo bem simples. Suponha que o departamento de marketing veio com o requisito que devemos aplicar um certo desconto em um produto, baseado em algumas regras:

1. Quando o produto é Vermelho o desconto é de 10%.
2. Quando o produto é Verde o desconto é de 20%.
3. Quando o produto é Azul o desconto é de 30%.

Para isto, vamos criar uma classe chamada de Desconto. No VS Code, selecione o projeto principal (Loja) e clique no ícone *New File*. Dê o nome de *Desconto.cs* e pressione *Enter*. Quando o arquivo for criado, adicione o esqueleto da classe e nada mais.

```
using System;

namespace Loja
{
    public class Desconto
    {
    }
}
```

Salve o arquivo (CTRL+S). No nosso projeto de testes, selecione o arquivo *UnitTest1.cs*, pressione F2 para renomeá-lo e dê o nome de *DescontoTests.cs*.

Dentro do arquivo, troque o nome da classe UnitTest1 para DescontoTests também. O resultado final deve ficar assim:

```
using System;
using Xunit;

namespace Loja.Tests
{
    public class DescontoTests
    {
        [Fact]
        public void Test1()
        {
        }
    }
}
```

Agora é hora de criar nossa primeira funcionalidade: **Quando o produto é Vermelho, o desconto é de 10%.**

Seus testes podem ter qualquer nome, mas é uma boa prática que o nome seja o mais explícito possível para que um outro programador, ou mesmo você no futuro, saiba exatamente qual regra aquele teste cobre. Sendo assim, altere o nome de nosso primeiro teste de “Test1” para “Quando_o_produto_eh_vermelho_o_desconto_eh_de_10_porcento” e vamos escrever o teste:

```
[Fact]
public void Quando_o_produto_eh_vermelho_o_
desconto_eh_de_10_porcento()
{
    //Preparar
    var desconto = new Desconto();

    //Executar
    var valor = desconto.Calcule(Cor.Vermelho);
```

```
    //Verificar
    Assert.Equal(10, valor);
}
```

Algumas considerações aqui:

É comum dividir o teste em 3 etapas. Na etapa “Preparar”, vamos construir todos os objetos necessários para o nosso teste. Neste caso, temos somente o objeto “Desconto”.

Na fase “Executar”, nós chamamos o método que queremos testar. Aqui estamos testando o método “Calcule” da classe “Desconto”, passando como parâmetro a cor vermelha. Note que se você tentar compilar seu código agora (control+shift+b e enter), ele vai apresentar erros, pois nem este método e nem o enum com a cor vermelha ainda existem. Não se preocupe, nós já vamos consertar isso.

E por último a fase de “Verificar”. O xunit possui uma série de verificações pré-definidas presentes na classe estática “Assert”. Digite “Assert” e o ponto que você verá muitas opções presentes ali. Como queremos testar se nossa variável “valor” armazena o número 10, nós usamos o método “Equal”. Quando isto for executado, caso o valor não seja 10, o teste acusará o erro dizendo que falhou. Caso seja, é mostrado que ele passou.

Agora que sabemos a ordem de execução, vamos implementar estes métodos que estão dando erro. A ideia de escrever testes é esta mesma, a gente escreve o que quer testar sem se preocupar muito se os métodos existem ou não e depois quando chegarmos a uma interface legal, implementamos o código para que o teste passe. Desta forma, criamos nossos métodos com a perspectiva de quem os está consumindo, melhorando assim a clareza e facilidade de uso dos mesmos.

Para o método “Calcule”, coloque o cursor nele e aperte “CTRL+.”.

Um dropdown irá aparecer com a opção de gerar este método para você na classe “Desconto”. Aperte Enter depois.

```
[Fact]
0 references | Run Test | Debug Test
public void Quando_o_produto_eh_vermelho_o_desconto_eh_de_10_porcento()
{
    //Preparar
    var desconto = new Desconto();

    //Executar                      control+.              aperte enter
    var valor = desconto.Calcule(Cor.Vermelho);
                          Generate method 'Desconto.Calcule'
    //Verificar
    Assert.Equal(10, valor);
}
```

Figura 61 – Gerar o método

Pressione F12 em cima deste método para mostrar a classe Desconto, altere-a para que fique assim:

```
using System;

namespace Loja
{
    public class Desconto
    {
        public int Calcule(Cor cor)
        {
                throw new NotImplementedException();
        }
    }
}
```

E para finalizar, vamos criar nossa enum de cores. Crie um novo arquivo chamado “Cor.cs” e adicione este código:

```
namespace Loja
{
    public enum Cor
    {
        Vermelho
    }
}
```

Não se preocupe com as outras cores agora, vamos criar o código estritamente necessário para o teste passar e devagar vamos incrementando. Compile o código para ter certeza que não há qualquer erro e vamos rodar o primeiro teste.

**Executando os Testes de Unidade**

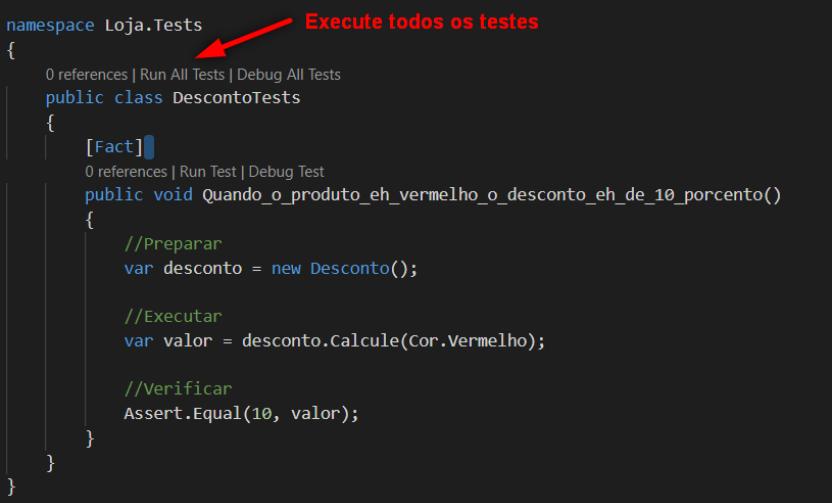
Há duas maneiras de rodar seus testes: via linha de comando ou de dentro do VSCode. Para linha de comando, vá no terminal, no diretório de seu projeto (no meu caso, C:\dev\csharp-com-testes\) e digite:

```
dotnet test
```

O dotnet vai compilar seu código, verificar que existe um projeto de testes em algum subdiretório e rodar todos os testes encontrados. Executando este comando neste momento você receberá uma mensagem que o teste falhou:

```
xUnit.net 00:00:00.41] Loja.Tests.DescontoTests.Quando_o_produto_eh_vermelho_o_desconto_eh_de_10_porcento [FAIL]
    Failed Loja.Tests.DescontoTests.Quando_o_produto_eh_vermelho_o_desconto_eh_de_10_porcento [2 ms]
  Error Message:
    System.NotImplementedException : The method or operation is not implemented.
  Stack Trace:
     at Loja.Desconto.Calcule(Cor cor) in C:\dev\csharp-
```

```
com-testes\Loja\Desconto.cs:line 9
        at  Loja.Tests.DescontoTests.Quando_o_produto_eh_vermelho_o_desconto_eh_de_10_porcento()  in  c:\dev\csharp-com-testes\Loja.Tests\DescontoTests.cs:line 15

    Failed!  - Failed:    1, Passed:    0, Skipped:    0, Total:    1, Duration: 2 ms - Loja.Tests.dll (net5.0)
```

O motivo da falha a gente sabe, o método Calcule ainda não possui uma implementação. Esta etapa é importante, queremos sempre ter um teste que falhe primeiro, para sabermos que nossa implementação posterior resolve o problema do teste e o faz passar.

Vá na classe Desconto e implemente o mínimo para fazer o teste passar. Poderia ser algo como:

```
public class Desconto
{
    public int Calcule(Cor cor) => 10;
}
```

Pronto, vamos rodar o teste novamente para ver se isto resolve, mas desta vez, faça pelo VS Code. Abra a classe *DescontoTests* e como o VS Code já detecta que esta classe tem testes, você vai encontrar uma opção *Run All Tests* bem acima do nome da classe, basta clicar lá:

```
namespace Loja.Tests                    Execute todos os testes
{
    0 references | Run All Tests | Debug All Tests
    public class DescontoTests
    {
        [Fact]
        0 references | Run Test | Debug Test
        public void Quando_o_produto_eh_vermelho_o_desconto_eh_de_10_porcento()
        {
            //Preparar
            var desconto = new Desconto();

            //Executar
            var valor = desconto.Calcule(Cor.Vermelho);

            //Verificar
            Assert.Equal(10, valor);
        }
    }
}
```

Figura 62 - Classe de testes

O teste deve passar desta vez e você verá a confirmação na janela de output:

```
—Test Execution Summary—

Loja.Tests.DescontoTests.Quando_o_produto_eh_vermelho_o_desconto_eh_de_10_porcento:
    Outcome: Passed
Total tests: 1. Passed: 1. Failed: 0. Skipped: 0
```

Regra número 1 implementada, agora vamos a número 2: **Quando o produto é Verde, o desconto é de 20%**.

Comece implementando um segundo teste, no mesmo arquivo DescontoTest.cs:

```
[Fact]
public void Quando_o_produto_eh_verde_o_desconto_eh_de_20_porcento()
{
```

```
    //Preparar
    var desconto = new Desconto();

    //Executar
    var valor = desconto.Calcule(Cor.Verde);

    //Verificar
    Assert.Equal(20, valor);
}
```

Lembre-se que a cor Verde ainda não existe, adicione-a ao enum para que o código compile, ficando assim:

```
public enum Cor
{
    Vermelho,
    Verde
}
```

Rode todos os testes novamente, o resultado será:

```
— Test Execution Summary —

Loja.Tests.DescontoTests.Quando_o_produto_eh_verme-
lho_o_desconto_eh_de_10_porcento:
    Outcome: Passed
Loja.Tests.DescontoTests.Quando_o_produto_eh_verde_o_
desconto_eh_de_20_porcento:
    Outcome: Failed
    Error Message:
    Assert.Equal() Failure
Expected: 20
Actual:   10
    Stack Trace:
      at Loja.Tests.DescontoTests.Quando_o_produto_eh_ver-
de_o_desconto_eh_de_20_porcento() in c:\dev\csharp-com-tes-
tes\Loja.Tests\DescontoTests.cs:line 31
```

```
Total tests: 2. Passed: 1. Failed: 1. Skipped: 0
```

Nosso primeiro teste passa e o segundo falha. Vamos arrumar o código para fazer tudo passar:

```
public class Desconto
{
public int Calcule(Cor cor) => cor == Cor.Verde ? 20 : 10;
}
```

Rodando os testes agora:

```
— Test Execution Summary —

Loja.Tests.DescontoTests.Quando_o_produto_eh_vermelho_o_desconto_eh_de_10_porcento:
    Outcome: Passed
Loja.Tests.DescontoTests.Quando_o_produto_eh_verde_o_desconto_eh_de_20_porcento:
    Outcome: Passed
Total tests: 2. Passed: 2. Failed: 0. Skipped: 0
```

Regra número 1 e 2 implementadas, agora vamos a número 3: **Quando o produto é Azul, o desconto é de 30%**.

Como sempre, implementamos primeiro um teste que falhe, para depois criar o código que o faz passar:

```
[Fact]
public void Quando_o_produto_eh_azul_o_desconto_eh_de_30_porcento()
{
    //Preparar
    var desconto = new Desconto();

    //Executar
    var valor = desconto.Calcule(Cor.Azul);
```

```
    //Verificar
    Assert.Equal(30, valor);
}
```

Adicione a cor “Azul” a enum e rode os testes:

```
— Test Execution Summary —

Loja.Tests.DescontoTests.Quando_o_produto_eh_vermelho_o_desconto_eh_de_10_porcento:
    Outcome: Passed
Loja.Tests.DescontoTests.Quando_o_produto_eh_azul_o_desconto_eh_de_30_porcento:
    Outcome: Failed
    Error Message:
    Assert.Equal() Failure
Expected: 30
Actual:   10
    Stack Trace: at Loja.Tests.DescontoTests.Quando_o_produto_eh_azul_o_desconto_eh_de_30_porcento() in c:\dev\csharp-com-testes\Loja.Tests\DescontoTests.cs:line 44

Loja.Tests.DescontoTests.Quando_o_produto_eh_verde_o_desconto_eh_de_20_porcento:
    Outcome: Passed
Total tests: 3. Passed: 2. Failed: 1. Skipped: 0
```

Novamente o teste falhou. A nova implementação fica:

```
public class Desconto
{
    public int Calcule(Cor cor) => cor switch
    {
        Cor.Vermelho => 10,
        Cor.Verde => 20,
```

```
        _ => 30
    };
}
```

Agora basta rodar os testes novamente e pronto:

```
----- Test Execution Summary -----

Loja.Tests.DescontoTests.Quando_o_produto_eh_vermelho_o_
desconto_eh_de_10_porcento:
    Outcome: Passed
Loja.Tests.DescontoTests.Quando_o_produto_eh_azul_o_
desconto_eh_de_30_porcento:
    Outcome: Passed
Loja.Tests.DescontoTests.Quando_o_produto_eh_verde_o_
desconto_eh_de_20_porcento:
    Outcome: Passed
Total tests: 3. Passed: 3. Failed: 0. Skipped: 0
```

**Adicionando mais um requisito**

Vamos supor que alguns meses depois, o pessoal de Produto solicita para que seja adicionada a cor Laranja. Ir direto apenas adicionando a nova cor ao enum parece uma solução simples, mas fazer isso gera um comportamento indesejado. Sendo assim, vamos criar um teste para ver como essa nova cor impacta os descontos:

Nossa enum vai ficar assim:

```
public enum Cor
{
    Vermelho,
    Verde,
    Azul,
    Laranja
}
```

E como não houve nenhuma menção ao desconto, este deve ser zero:

```
[Fact]
public void Quando_o_produto_eh_laranja_nao_ha_
desconto()
{
    //Preparar
    var desconto = new Desconto();

    //Executar
    var valor = desconto.Calcule(Cor.Laranja);

    //Verificar
    Assert.Equal(0, valor);
}
```

Ao rodar todos os testes temos:

```
----- Test Execution Summary -----

Loja.Tests.DescontoTests.Quando_o_produto_eh_vermelho_o_
desconto_eh_de_10_porcento:
    Outcome: Passed
Loja.Tests.DescontoTests.Quando_o_produto_eh_laranja_nao_
ha_desconto:
    Outcome: Failed
    Error Message:
    Assert.Equal() Failure
Expected: 0
Actual:   30
    Stack Trace:
      at Loja.Tests.DescontoTests.Quando_o_produto_eh_
```

```
laranja_nao_ha_desconto() in c:\dev\csharp-com-testes\Loja.
Tests\DescontoTests.cs:line 57

    Loja.Tests.DescontoTests.Quando_o_produto_eh_azul_o_
desconto_eh_de_30_porcento:
        Outcome: Passed
    Loja.Tests.DescontoTests.Quando_o_produto_eh_verde_o_
desconto_eh_de_20_porcento:
        Outcome: Passed
    Total tests: 4. Passed: 3. Failed: 1. Skipped: 0
```

Veja que o teste pegou o problema. A classe Desconto deve ser alterada para ficar protegida de todas as cores não especificadas:

```
public class Desconto
{
    public int Calcule(Cor cor) => cor switch
    {
        Cor.Vermelho => 10,
        Cor.Verde => 20,
        Cor.Azul => 30,
        _ => 0
    };
}
```

Nesta implementação do switch, o parâmetro cor é avaliado de acordo com a enum, se for Vermelho o valor é 10, se for Verde o valor é 20, se for Azul o valor é 30, e o caso contrário, o valor padrão será 0. E ponto, tudo passando:

```
    —Test Execution Summary —

    Loja.Tests.DescontoTests.Quando_o_produto_eh_vermelho_o_
desconto_eh_de_10_porcento:
        Outcome: Passed
    Loja.Tests.DescontoTests.Quando_o_produto_eh_laranja_nao_
ha_desconto:
```

```
        Outcome: Passed

    Loja.Tests.DescontoTests.Quando_o_produto_eh_azul_o_
desconto_eh_de_30_porcento:
        Outcome: Passed
    Loja.Tests.DescontoTests.Quando_o_produto_eh_verde_o_
desconto_eh_de_20_porcento:
        Outcome: Passed
    Total tests: 4. Passed: 4. Failed: 0. Skipped: 0
```

Trabalho completo. Seu código está pronto e muito bem testado.

## 10 VANTAGENS DE SE ESCREVER TESTES DE UNIDADE

Como você pode ver, escrever testes requer muito mais código do que somente implementar a funcionalidade, então você deve estar se perguntando, vale mesmo a pena?

### 1 - Segurança em fazer alterações de código

Quanto mais seu projeto crescer, mais complexo ele vai ficar, e mais difícil vai ser para um programador conhecer todas as suas linhas de código. Desta forma, o risco de alterar uma parte do sistema e causar um bug em outra aumenta também. Outro cenário é o caso de você ter acabado de entrar em um projeto e ter que fazer uma alteração. Como você vai saber quais efeitos colaterais esta mudança causará sem conhecer o sistema todo?

Os testes de unidade vão ser justamente essa rede de segurança. Você pode fazer sua alteração e depois rodar todos os testes para verificar que tudo ainda continua funcionando perfeitamente. Isto serve para trabalhar tanto em território desconhecido como quando você está criando um algoritmo complexo e precisa saber que cada pequeno avanço não está inadvertidamente quebrando em outro lugar.

**2 - Os testes são uma ótima documentação das regras de negócio**

Quando você começa a trabalhar com um sistema novo, o melhor lugar para entender como o sistema funciona e quais são suas regras é ler os testes de unidade.

Vamos dar uma olhada nos testes que criamos, por exemplo:

Quando_o_produto_eh_vermelho_o_desconto_eh_de_10_porcento()
Quando_o_produto_eh_verde_o_desconto_eh_de_20_porcento()
Quando_o_produto_eh_azul_o_desconto_eh_de_30_porcento()

Bastando ler o nome dos métodos, aprendemos na hora como o sistema deve se comportar. Isso acontece porque os testes tratam as funções como caixas pretas, ligando apenas para as entradas e as saídas, sendo assim sua leitura um perfeito resumo do comportamento do sistema.

**3- Testes são uma ótima forma de se aprender a usar uma biblioteca nova**

Como quem desenvolveu escreveu os testes com a visão de um usuário daquela biblioteca, você vai encontrar todos os casos de uso de suas funções por lá. Sempre que você tiver acesso ao código-fonte de algum pacote nuget que você for usar, dê uma olhada nos testes de unidade. Você vai aprender a usá-la muito mais rápido.

**4- Use os testes para começar mais rápido no dia seguinte**

Uma boa coisa a se fazer no final de um longo dia de programação, é escrever somente o enunciado do próximo teste que você quer implementar e salvar com ele falhando, sem nenhuma implementação. No outro dia quando você voltar, basta rodar todos os testes para saber exatamente em que ponto parou no dia anterior

e qual é sua próxima tarefa.

Mas atenção, nunca envie código com testes falhando para o sistema controle de versão! Seu time não vai gostar nem um pouco. Envie seus testes todos passando e só depois escreva este último falhando, deixando-o local até você fazê-lo passar.

**5- Escrever os testes antes de escrever a funcionalidade aumenta a qualidade do código**

Em nosso exemplo, sempre escrevemos primeiro o teste para somente depois escrever o código que faz passá-lo. Isto não é uma regra estrita, mas é uma boa prática. A vantagem de escrever o teste antes é garantir que todo código que você vai escrever vai estar testado.

Outra coisa também é que fazendo isso você é obrigado a escrever um código facilmente testável, ou seja, a prática vai guiando como você estrutura a arquitetura do seu sistema.

**6- Testes de Unidade são automatizados e podem ser executados em um servidor de build**

Não é só no seu computador que os testes devem ser executados. Toda vez que você envia seu código para um sistema de controle de versão, os testes devem ser rodados automaticamente para verificar que o que está indo para produção foi testado em uma máquina independente. Lembre-se que em nosso laptop nós temos muitas ferramentas e frameworks de desenvolvimento e com certeza esse ambiente não reflete onde o sistema final vai rodar. Desta forma, conseguimos pegar os problemas de ambiente muito antes do defeito chegar em nossos clientes.

**7- Testes de Unidade ajudam a trabalhar em equipe**

Na profissão de desenvolvedor de software, dificilmente vamos trabalhar sozinhos. Seu colega de trabalho vai estar alterando có-

digo no mesmo sistema que você e muitas vezes até no mesmo arquivo.

Se todo mundo escrever testes, quando esse código todo se juntar, os testes podem ser executados garantindo que o você mudou não quebra o código do seu colega e vice-versa.

**8- Refatoração fica muito mais simples**

O processo de refatoração é quando pegamos código que não está como queremos e fazemos mudanças para a melhoria de sua qualidade, sem alterar nenhuma funcionalidade. Sem os testes, fica muito difícil mudar grandes seções de código e garantir que aquilo vai continuar funcionando da mesma forma como antes. Já com uma boa cobertura de código, podemos mudar tudo à vontade e rodar os testes o tempo todo para averiguar se o funcionamento continua o mesmo. Testes de Unidade nos deixa muito mais confiantes.

**9- Feedback muito mais rápido**

Ao programar sem escrever testes, você fica muito dependente de usar o debugger o tempo todo. O único jeito de você saber se algo funciona é dar play na aplicação e ir passo a passo olhando o resultado. Com testes de unidade, basta rodar os mesmos e verificar na hora o resultado. O ciclo de feedback é muito mais rápido nos ajudando a manter o foco de forma a ficarmos muito mais produtivos.

**10- O mesmo bug nunca volta a acontecer**

Sempre que for encontrado um bug em sua aplicação, não vá diretamente resolver o problema. Ao invés disso, escreva um teste que reproduza o bug, fazendo o teste falhar. Arrume o defeito modificando o código até que o teste passe. Com este teste no lugar, aquele mesmo problema nunca voltará.

## QUANDO NÃO USAR TESTES DE UNIDADE?

Como toda prática, você deve utilizar testes de unidade somente quando isso vai te trazer benefícios. Na seção anterior, vimos 10 exemplos de retorno positivo ao se usar testes, mas podem haver situações onde não vale a pena.

Isto claramente vai variar de pessoa para pessoa e projeto a projeto, mas um bom exemplo seria quando você está experimentando com algum framework ou biblioteca nova e ainda não sabe bem o que quer com aquilo. A liberdade de escrever código sem pensar nas consequências vai te trazer rapidez e mais criatividade. Passado esse momento, você pode organizar seu projeto e então, reescrever tudo com testes se isso fizer sentido.

### Algumas Considerações

1 - Cada teste de unidade deve testar somente uma coisa. É comum cair na tentação de colocar vários "Asserts" no mesmo método, mas deixe esse número o mínimo possível. Seus testes devem ser simples e devem falhar por um único motivo, assim você sabe exatamente onde está o problema. Daí que vem o nome "Unidade".

2 - Deixe seus testes em uma biblioteca separada, somente com testes. Desta forma o código de testes é usado para testar, mas não é enviado junto com o produto final para produção. Outra coisa é deixar bem claro onde estão os testes e facilitar para que qualquer desenvolvedor os encontre facilmente.

3 - Trate seu código de testes como qualquer outro código. Os testes devem ter a mesma qualidade e cuidado que seu código de produção tem. Refatore-os, deixe-os melhores e mais fáceis de se ler à medida que forem evoluindo. É claro que você pode ter um bug no seu código de testes, mas se você fizer com cuidado a chance de se ter um bug nos testes e no código de produção ao mesmo tempo é muito menor.

4 - Testes de Unidade devem ser rápidos! Lembre-se que um dos maiores benefícios de rodar os testes é ter este feedback super rápido de como nosso código está funcionando. Com o passar do tempo, teremos centenas de testes em nossa aplicação e se eles demorarem muito para rodar, provavelmente vamos parar de fazê-lo o todo momento, perdendo seus benefícios. Sendo assim, refatore seus testes constantemente começando sempre pelos mais lentos.

5 - Testes de Unidade não são Testes de Integração. Os testes de unidade devem ser totalmente portáteis e testar somente seu código local. Se a chamada que você está fazendo termina em outro sistema, um banco de dados ou mesmo um arquivo externo, você deve alterar seu código para isolar estas chamadas utilizando interfaces. Desta forma, você pode ter uma implementação desta interface que realmente acessa o banco de dados quando seu código está rodando em produção e outra que apenas simula o banco para fins de testes. Não vamos explorar este cenário neste livro, mas pesquise sobre “mocks” e “stubs” para saber mais sobre o assunto.

Escrever Testes de Unidade não é uma tarefa trivial e exige esforço adicional do programador. Por outro lado, a prática traz muitos benefícios, principalmente no aumento de qualidade e para equipes trabalhando juntas. Por este motivo, as principais empresas de software utilizam e valorizam o programador que domina esta técnica.

# ANEXO – CÓDIGOS FONTES DO LIVRO

Durante o livro mostramos vários exemplos de códigos para que você pudesse entender a linguagem C# e suas diversas funcionalidades. Todos estes códigos estão no GitHub deste livro no endereço: https://github.com/MVPsbr/livrocsharp. E para utilizá-los em seu aprendizado baixe os códigos fontes de acordo com as instruções abaixo.

**BAIXANDO O CÓDIGO FONTE DOS EXEMPLOS DO LIVRO**

O nosso repositório de código está no GitHub e usamos o formato Git, pois ele é amplamente utilizado em diversos tipos de projetos e empresas.

Primeiramente instale o Git no seu computador baixando os arquivos no site https://git-scm.com/downloads. Existem versões para várias plataformas, então baixe a que mais de adeque ao seu cenário.

Depois de instalado, você precisará clonar o nosso repositório de código, o que significa basicamente, copiar o código do site do GitHub para o seu computador. Para isto, use o comando:

git clone https://github.com/MVPsbr/livrocsharp.git

Este é o endereço do repositório no GitHub: https://github.com/

MVPsbr/livrocsharp

E para fazer o clone você usa este endereço: https://github.com/MVPsbr/livrocsharp.git

Você também pode clonar o repositório pelo Visual Studio Code ou pelo Visual Studio.

## CLONANDO OS EXEMPLOS PELO VISUAL STUDIO

Para clonar o repositório usando o Visual Studio, você escolhe a opção de acordo com a figura a seguir logo ao iniciar a ferramenta:

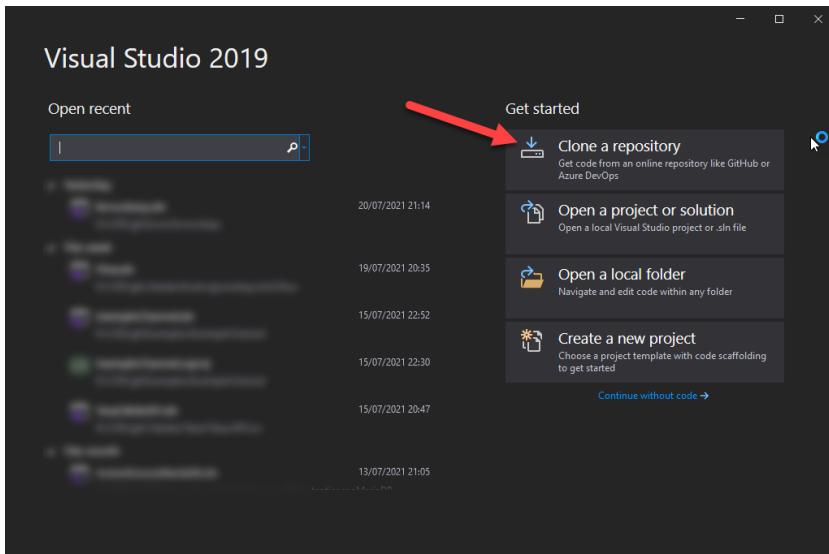



Figura 63 - Clonando o repositório pelo Visual Studio

Depois coloque a URL do repositório e o diretório para onde quer copiar no seu computador, de acordo com a figura a seguir:

Figura 64 - Clonando o repositório pelo Visual Studio

## CLONANDO OS EXEMPLOS PELO VISUAL STUDIO CODE

No Visual Studio Code você clica no ícone do Git e depois em "Clone Repository" de acordo com a figura a seguir:

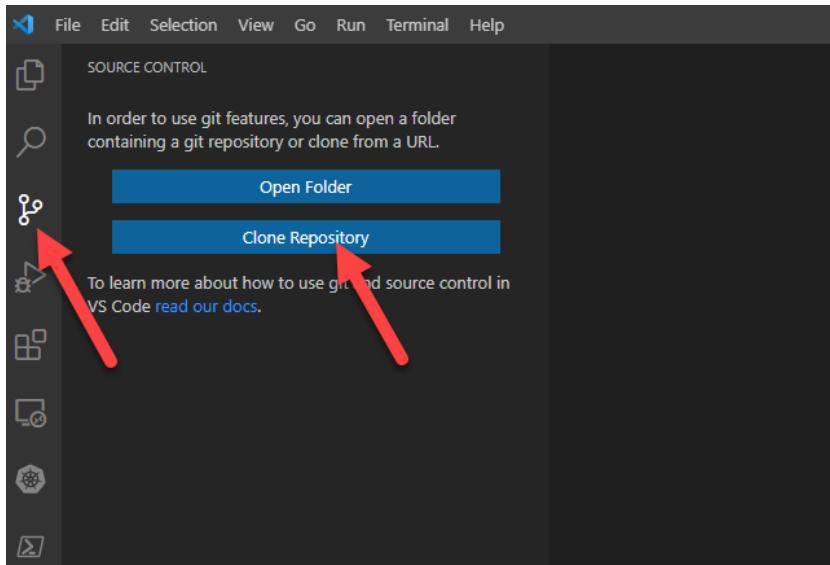



Figura 65 - Clonando o pelo Visual Studio Code.

Depois coloque a URL do repositório de acordo com a figura a seguir:

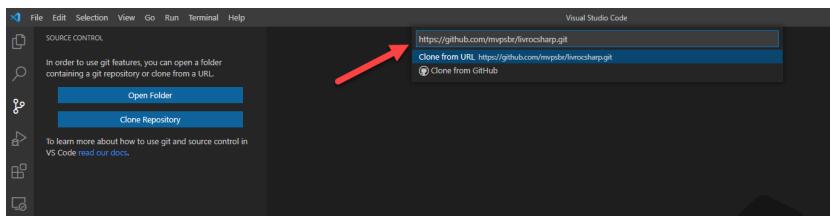



Figura 66 - Clonando o pelo Visual Studio Code.

## USANDO O PROJETO DE EXEMPLO

Depois de clonar o repositório do livro, você terá acesso a todos os exemplos, divididos por capítulos, que vão lhe ajudar a assimilar todos os conhecimentos transmitidos pelo livro.

## DIFERENÇAS ENTRE O VISUAL STUDIO E O VISUAL STUDIO CODE

O Visual Studio trabalha com o conceito de Solution (arquivo .SLN), então você deve abrir este arquivo para trabalhar com o projeto exemplo. Já o Visual Studio Code, trabalha com o conceito de pastas e você precisa abrir cada pasta individualmente para testar os exemplos. O Visual Studio também pode trabalhar com o conceito de pastas.

## UMA DICA IMPORTANTE

Se você não está familiarizado com o uso do git, procure aprender o quanto antes, pois a maioria das empresas tem adotado este tipo de repositório para guardar seus códigos fontes.